CodeWarrior®

Pascal Language Reference

Because of last-minute changes to CodeWarrior, some of the information in this manual may be inaccurate. Please read the Release Notes on the CodeWarrior CD for the latest up-to-date information.

Revised: 98/08/17 map

Metrowerks CodeWarrior copyright ©1993–1998 by Metrowerks Inc. and its licensors. All rights reserved.

Documentation stored on the compact disk(s) may be printed by licensee for personal use. Except for the foregoing, no part of this documentation may be reproduced or transmitted in any form by any means, electronic or mechanical, including photocopying, recording, or any information storage and retrieval system, without permission in writing from Metrowerks Inc.

Metrowerks, the Metrowerks logo, CodeWarrior, and Software at Work are registered trademarks of Metrowerks Inc. PowerPlant and PowerPlant Constructor are trademarks of Metrowerks Inc.

All other trademarks and registered trademarks are the property of their respective owners.

ALL SOFTWARE AND DOCUMENTATION ON THE COMPACT DISK(S) ARE SUBJECT TO THE LICENSE AGREEMENT IN THE CD BOOKLET.

How to Contact Metrowerks:

U.S.A. and international	Metrowerks Corporation P.O. Box 334 Austin, TX 78758 U.S.A.
Canada	Metrowerks Inc. 1500 du College, Suite 300 Ville St-Laurent, QC Canada H4L 5G6
Ordering	Voice: (800) 377–5416 Fax: (512) 873–4901
World Wide Web	`http://www.metrowerks.com`
Registration information	`register@metrowerks.com`
Technical support	`support@metrowerks.com`
Sales, marketing, & licensing	`sales@metrowerks.com`
CompuServe	Go: `Metrowerks`

Table of Contents

1 Introduction **13**

Welcome to CodeWarrior Pascal 13
Read the Release Notes 14
What's New 14
Starting Points 14
Structure of this Manual 16
Conventions Used In This Manual 16
Where to Learn More 19

2 Other Pascals **23**

Other Pascals Overview 23
CodeWarrior Pascal and ANS Pascal 23
Exceptions to ANS Pascal 24
Extensions to ANS Pascal 24
Implementation-Dependent Features 26
CodeWarrior Pascal and MPW Pascal 27
Exceptions to MPW Pascal. 27
Extensions to MPW Pascal. 27
CodeWarrior Pascal and Turbo Pascal 28
Support for Turbo Pascal Language Extensions. 29

3 Symbols **31**

Symbols Overview 31
Special Symbols And Reserved Words 32
Identifiers 33
Numbers 34
Labels. 36
Quoted String Constants. 36
Quoted Character Constants 37
Delimiters 38
Comments. 38
Compiler Directives. 39

4 Data Types 41

Data Types Overview 41
Type Declarations 42
Real Types 43
Scalar Types 44
INTEGER 46
LONGINT 46
UNSIGNEDBYTE 48
UNSIGNEDWORD 48
UNSIGNEDLONG 48
BOOLEAN 48
CHAR 49
COMP 49
Enumerated Types 49
Subrange Types 50
String Types 51
Pascal Strings 51
C Strings 53
Pointer Types 53
Procedure And Function Types 54
Structured Types 55
About Structured Types 56
Array Types 56
PACKED ARRAY OF CHAR 58
Record Types 58
Set Types 61
File Types 62
Object Types 63
Opaque Types 64
Schemata 64
Schemata Overview 64
Schemata Limitations 66
Type Compatibility 66
Same Type Compatibility 67
Simple Compatibility 67

Assignment Compatibility 68
Type Coercion 69
User-Defined Anonymous Types 70

5 Constants and Variables 73
Constants and Variables Overview 73
Constants 73
Constant Declarations. 74
Predefined Constants 75
Variables 76
Variable Declarations 76
Variable Initialization 78
Variable Accesses. 80
Qualifiers to Variable Accesses 81

6 Expressions 87
Expressions Overview. 87
Atomic Expressions 88
Compound Expressions 89
Operators and Operands 89
Operator Precedence 90
Constant Compound Expressions 91
Expression Syntax 91
Operators 93
Arithmetic Operators 94
Boolean Operators 96
String Operators 98
Set Operators 98
Relational Operators 99
The @ Operator 101
The Set Constructor. 102

7 Statements 105
Statements Overview 105
Repetition Statements 106
Conditional Statements 107

Control Statements 108
Individual Statement Descriptions 109
The Assignment Statement 109
The Compound Statement 111
The Procedure Statement 112
The WITH Statement 113
The FOR Statement 114
The WHILE Statement 116
The REPEAT Statement 117
The IF Statement 118
The CASE Statement 120
The GOTO Statement 121
The RETURN Statement 122
The CYCLE Statement 122
The LEAVE Statement 123
The Null Statement 124

8 Procedures & Functions 125

Procedures and Functions Overview 125
Declaring Procedures and Functions 125
Procedure Declarations 126
Function Declarations 127
Procedure And Function Directives 129
Parameters 133
Value Parameters 135
Variable Parameters 136
Constant Parameters 137
Typeless Parameters 138
Procedural Parameters 138
Functional Parameters 141
Open Area Parameters 141
UNIV Parameters 142
The Ellipsis Parameter (...) 143
Parameter List Compatibility 143
Invoking Procedures and Functions 144

Invoking Procedures 144
Invoking Functions 145
Using C Routines Within Pascal 146

9 Object Pascal 147
Object Pascal Overview 147
Object Oriented Programming Basics 148
Procedural Programming and Object Oriented Programming 148
Definitions 149
Object Pascal Basics 149
Creating Objects 152
Declaring Object Types 152
Object and Member Scope 153
Object Type Membership 154
Declaring Object Instances 155
Object Reference Variables 155
Declaring Methods 156
Calling Methods 158

10 Blocks and Scope 161
Blocks and Scope Overview 161
Block Structure 161
Scope Rules 164
Exceptions to the General Scope Rules 164

11 Programs and Units 169
Programs and Units Overview 169
Program Structure 170
Unit Structure 170
Declaration Scope within Programs and Units 173
The USES Clause 174
Units That Use Other Units 176
Circular References 177

12 Files and I/O 181
Files and I/O Overview 181

Pascal File Variables 182
File Variable Types 182
Predeclared File Variables 184
The File Window Variable 185
Using Pascal Files 186
Opening and Closing Files 186
Sequential Versus Random Access 186

13 Syntax Summary 189

Syntax Summary Overview 189
Metrowerks Pascal Syntax Specification 189
actual parameter 189
actual parameter list 189
array type 189
assignment statement 190
atomic expression 190
base type 190
block 190
case 190
CASE statement 190
compound statement 190
constant 191
constant atomic expression 191
constant declaration 191
constant declaration part 191
constant expression 191
constant factor 191
constant identifier 192
constant range 192
constant term 192
control variable 192
CYCLE statement 192
digit 192
digit sequence 192
enumerated type 193

expression 193
factor 193
field declaration 193
field designator 193
field list 193
file buffer symbol 193
file type 194
final value 194
fixed part 194
FOR statement 194
formal parameter list 194
function body 194
function call 194
function declaration 195
function heading 195
function identifier 195
function type 195
GOTO statement 195
hex digit 195
hex digit sequence 195
identifier 196
identifier list 196
IF statement 196
implementation part 196
index 196
index type 196
initial value 197
interface part 197
interface type declaration 197
interface type declaration part 197
label 197
label declaration part 197
LEAVE statement. 197
member group 198
method declaration 198

method heading 198
null statement 198
object data heading 198
object data field scope 198
object type 199
object type identifier 199
opaque type 199
opaque type identifier 199
ordinal type identifier 199
otherwise clause 199
parameter declaration 199
pointer symbol 200
pointer type 200
pointer type identifier 200
procedure & function declaration part 200
procedure body 200
procedure declaration 200
procedure heading 201
procedure identifier 201
procedure statement 201
procedure type 201
program 201
program heading 201
program parameters 201
qualifier 202
qualifier identification 202
quoted character constant 202
quoted string constant 202
real type 202
real type identifier 202
real-type numbers 202
record type 203
REPEAT statement 203
RETURN statement 203
scalar type 203

schema declaration 203
schema definition 203
schema type 204
set constructor 204
set type 204
sign 204
simple constant expression 204
simple expression 204
simple type 204
size attribute. 205
statement 205
statement part 205
step value 205
string character 205
string type. 205
string type identifier 206
structured type. 206
stuctured type identifier. 206
subrange type 206
tag field type 206
term 206
type 206
type declaration 207
type declaration part 207
type identifier 207
underscore 207
unit 207
unit heading 207
unsigned constant 208
unsigned number 208
USES clause 208
variable access 208
variable declaration. 208
variable declaration part 208
variable identifier 208

variable initialization 209
variant 209
variant part 209
WHILE statement 209
WITH statement 209

1 Introduction

Welcome to the CodeWarrior Pascal programming language. This manual is a complete reference to the structure and usage of this language.

Welcome to CodeWarrior Pascal

CodeWarrior Pascal is a variant of the Pascal programming language. It supports ANS ("standard") Pascal, contains many of the features of other popular Pascals, plus several extensions.

The goal of this manual is to present a strictly *implementation-independent* view of the CodeWarrior Pascal language. What this means is that we purposefully leave out of this manual details about how the compiler actually translates Pascal code into machine-level code which will run on one platform or another. In this manner, we can present a high-level description of the language, a description that will be compatible with whatever platform you happen to be generating code for.

The *Pascal Compilers Reference* is an important companion to this manual. It is dedicated specifically to the implementation and platform-specific aspects of the CodeWarrior Pascal compiler not covered by this reference. Another manual, the *Pascal Library Reference*, covers the built-in routines in CodeWarrior Pascal.

This introductory chapter covers these topics:

- Read the Release Notes—a reminder that up-to-the-minute information can be found in the CodeWarrior release notes
- What's New—changes since the last version of this document
- Starting Points—find out how to best use this manual depending on your experience and background with Pascal and CodeWarrior

- Structure of this Manual—how the information in this manual is organized
- Conventions Used In This Manual—the typographic and notational conventions used throughout the manual
- Where to Learn More—sources of information that may help you to learn more about Pascal programming and using the CodeWarrior IDE

Read the Release Notes

Before using CodeWarrior Pascal, please take the time to read the release notes in the `Release Notes` folder. They contain important information about new features, bug fixes, and any late-breaking changes that may have occurred *after* the production of this manual.

What's New

This revision of the *Pascal Language Reference* contains this new and updated information:

- "Declaring Object Types" on page 152—updated to show the syntax of declaring object types
- "Object and Member Scope" on page 153—updated information on the `PRIVATE`, `PROTECTED`, and `PUBLIC` keywords

Starting Points

Don't know where to begin? This section helps you to figure out a few good starting points based on your knowledge of and experience with Pascal and the CodeWarrior environment.

Everyone

If you are trying to figure out something concerning how to build or set up your Pascal Project in CodeWarrior, don't start here. The *CodeWarrior IDE User Guide* contains all the information about setting up and modifying your project. If you want to modify settings of the Pascal compiler itself, see the *Pascal Compilers Reference* for de-

tails on how to do it. To find out how to use the Pascal compiler to generate software for a particular operating system or processor, see its corresponding *Targeting* manual. For example, to learn how to use CodeWarrior Pascal to write a program to run on a Mac OS computer, see *Targeting Mac OS*.

Beginners

Those without a prior knowledge of Pascal can safely skip the chapter, "Other Pascals Overview" on page 23. This chapter basically details the differences between CodeWarrior Pascal and other variations of the language. You will instead want to go right to the second chapter, which is the first of the chapters that explain fundamental Pascal language concepts. If you ever want to switch from CodeWarrior Pascal to some other version of Pascal, you may want to go back and read this chapter to see how they differ.

If you are new not only to Pascal but to programming in general, see the "Where to Learn More" section below for suggestions about good introductory books using Pascal as the language of instruction.

Pascal programmers

Those who have experience with Pascal but have not used CodeWarrior Pascal will find the "Other Pascals Overview" chapter useful in finding out where to watch out for differences between CodeWarrior Pascal and the version of Pascal that you happen to be used to. Specifically, it covers ANS Standard Pascal, Apple's MPW Pascal, and Turbo Pascal.

Pascal experts

In addition to reading the "Other Pascals Overview" chapter, Pascal programmers who want *just the facts* can go directly to the "Syntax Summary" Appendix which is a complete and compact specification of the syntax of CodeWarrior Pascal. There is no explanation in the syntax summary; it is nothing more than a compilation of the all syntax diagrams used throughout the rest of the manual.

Structure of this Manual

Each chapter in this manual discusses, in depth, a concept or group of concepts of the Pascal language. The only exception to this is the "Other Pascals Overview" chapter, which is meant for those already familiar with the Pascal programming language; it cuts across all aspects of the language, noting the differences between CodeWarrior Pascal and other common Pascal variants.

Every chapter starts out with a chapter "Overview" section which gives a little background on the concepts to be presented along with a short description of how the chapter is laid out.

Many concepts require a description of the associated syntax, in other words how the abstract concept is actually written in CodeWarrior Pascal code. All such syntax is presented in a diagrammatic way which is described later in this chapter (see "Syntax Diagrams" on page 17.) The "Syntax Summary Overview" on page 189 is an alphabetized collection of all the syntax diagrams that appear throughout the manual. This appendix serves as a definitive specification of the syntax of CodeWarrior Pascal.

Conventions Used In This Manual

Typographic Conventions

This manual uses typographic conventions to distinguish between kinds of words and symbols:

- Ordinary text is in plain letters, like the kind you are reading now.
- Special technical terms are printed *this way* when they are first defined. After that, they are treated as ordinary text.
- Elements of the Pascal language (or any other programming language) are printed `this way`. This helps you avoid confusing them with ordinary text.
- Artificial terms, which have meaning only in this book, are printed *this way*. These terms are used primarily to indicate

parts of programming syntax that you replace with your own symbols.

- Menu commands, dialog box controls, and other user interface items are printed **this way** when they are first mentioned. After that, they are treated as ordinary text.

Source Code Conventions

The following conventions for Pascal source code are used in this manual:

- Reserved words are printed in `ALL CAPITALS`.
- The names of procedures and functions are printed in `Initial Capitals`.
- The names of data types, constants, and variables are printed in `lowercase`.
- A sequence of two dots (..) indicates a range between two values. For example, 0 .. 127 means "0 through 127."

Syntax Diagrams

Throughout this manual, syntactic details about the language are specified with a kind of textual diagramming which serves to illustrate how the elements of the language are combined and put together to form correct Pascal source code.

The notation used in our diagrams is called "Extended Backus-Naur Form", or EBNF, and is a general-purpose language used for specifying computer language syntax. The idea is that each *abstract* element of syntax, all the way up to the *program* itself, can be described as a string of less and less abstract elements until you eventually end up with *terminal elements*, which are the actual string symbols which make up Pascal source code.

As an example, look at the syntax diagram for the *assignment statement* construction:

<assignment statement> ::=

(<variable access> | <function identifier>) **:=** <expression>

A word or words appearing between angled brackets ("<>") comprise an abstract (non-terminal) syntax element, so "assignment statement", "variable access", "function identifier", and "expression" are all the abstract syntax elements that we will be dealing with in this construction.

The "::=" symbol is used for defining an abstract syntax element in terms of other syntax elements. This is a *construction*. In this case, we are defining the assignment statement construction.

Terminal elements, or syntax elements which are actual symbols in Pascal source code, are printed like **`this`**. In the assignment statement example, the only terminal element is the assignment symbol, "**`:=`**".

Parentheses are used, just as they are in mathematical expressions, to isolate something to be treated as its own distinct unit. The "|" symbol is used, as it is in some computer languages, to signify a form of "or." In the case of EBNF, the "|" symbol implies mutual exclusion… it signifies that exactly one of the elements in a group separated by "|" symbols is allowed.

Now you know enough EBNF to understand the assignment statement syntax. It says that an assignment statement consists of either a variable access or a function identifier, followed by the "**`:=`**" symbol, followed by an expression. If we look at an actual assignment statement, we can see how this works:

```
aVariableAccess := an + expression
```

In this assignment statement, `aVariableAccess` represents, not surprisingly, the "<variable access>" element in the assignment statement syntax, while "`an + expression`" represents the "<expression>" part.

Here is a list of all the symbols used in EBNF notation, along with their meanings:

- Words enclosed in angled-brackets "<>" denote an abstract syntax object.
- "::=" defines an abstract syntax object. The object being defined is on the left and the definition is on the right.

- Strings of symbols within parentheses are treated as a single object.
- Strings of symbols within square brackets "[]" are optional; they can be included either once or not at all.
- Strings of symbols within braces "{}" can be included any number of times in a row, or they can not be included at all. "{**a**} **b**" for example, could be expressed as "`ab`", "`aaaaab`", or simply "`b`".
- Items separated by a vertical bar "`|`" are mutually exclusive; one and only one out of the group must be included.

Where to Learn More

The manual you are reading is fairly narrow in scope; there are lots of questions you might have about developing your Pascal program that aren't answered here. The purpose of this section is to give you a few suggestions about good places to look for answers to these questions.

CodeWarrior documentation

If you're looking for details about the Pascal compiler itself, including how to configure it and how it implements Pascal on the platform you are targeting, see the *Pascal Compilers Reference*. If you're looking for details about Pascal functions and procedures within the standard Pascal runtime or I/O libraries, or if you want to find out about using ANSI C within CodeWarrior Pascal, see the *Pascal Library Reference*.

There are several other CodeWarrior manuals which may be helpful to you even though they aren't specific to Pascal. The *CodeWarrior IDE User Guide* contains lots of information about how to use the CodeWarrior IDE, as well as all the information you'll need about how to set up and configure your Pascal project. The *Error Reference* explains and offers strategies for overcoming the errors issued by CodeWarrior compilers and linkers, including CodeWarrior Pascal. The *C Compilers Reference* will be important if you plan on using C or C++ code in your Pascal project.

To find out how to use the Pascal compiler to generate software for a particular operating system or processor, see its corresponding *Targeting* manual. For example, to learn how to use CodeWarrior Pascal to write a program to run on a Mac OS computer, see *Targeting Mac OS*

Other books and products

If you're new to programming and are looking for books that teach Pascal as well as the fundamental topics of programming, Discover Programming™ software covers everything the beginning programmer would need to get started programming using CodeWarrior.

A great non-introductory resource for Macintosh programming (entirely in Pascal!) is:

- Apple Computer, Inc., *Inside Macintosh* volumes, Addison-Wesley, 1994. These manuals are available as printed books, on CD-ROM, and on-line through Apple's World Wide Web and FTP sites on the Internet.

If you want to learn more about Object-Oriented methodology, here are a couple of suggestions among the large number of resources on the subject:

- Erich Gamma et al., *Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
- Wolfgang Pree, *Design Patterns for Object-Oriented Software Development*, Addison-Wesley, 1994.

A few other important books in the Pascal world are:

- Doug Cooper and Michael Clancy, *Oh! Pascal!*, W.W.Norton & Company, 1993.
- Kathleen Jensen and Niklaus Wirth, *Pascal User Manual and Report*, 3d edition, Springer-Verlag, 1985. Revised by Andrew B. Mickel and James F. Miner. The original definition of Pascal.
- Henry Ledgard, *The American Pascal Standard, With Annotations*, Springer-Verlag, 1984. An annotated guide to ANS Pascal, as defined by the American National Standards Institute.

Newsgroups

Internet *Usenet* newsgroups are great sources of help and up-to-the-minute information. There are several newsgroups which discuss issues pertaining to Pascal programming:

- `comp.lang.pascal.ansi-iso`: Pascal according to ANSI and ISO standards.
- `comp.lang.pascal.borland`: Borland's Pascals (including Turbo Pascal but not Delphi.)
- `comp.lang.pascal.mac`: Macintosh-based Pascals.
- `comp.lang.pascal.misc`: Pascal in general and ungrouped Pascals.

2

Other Pascals

This chapter contains brief descriptions of the differences between CodeWarrior Pascal and other implementations of the Pascal language. It also gives tips on porting source code from other Pascals to CodeWarrior Pascal.

Other Pascals Overview

While all flavors are Pascal are similar, each has its own subtle variations. This chapter describes the differences between CodeWarrior Pascal and several other popular Pascal packages.

The chapter covers the following topics:

- CodeWarrior Pascal and ANS Pascal
- CodeWarrior Pascal and MPW Pascal
- CodeWarrior Pascal and Turbo Pascal

CodeWarrior Pascal and ANS Pascal

Unless the CodeWarrior Pascal compiler is specifically set to comply with the guidelines of American National Standard (ANS) Pascal, it allows several exceptions and extensions to this standard, as described below. For information on how to set compiler options like ANS conformance, see the *Pascal Compilers Reference*.

The rest of this section categorizes the differences between ANS and CodeWarrior Pascal as follows:

- Exceptions to ANS Pascal
- Extensions to ANS Pascal
- Implementation-Dependent Features

Exceptions to ANS Pascal

The CodeWarrior Pascal compiler complies with the requirements of ANSI/IEEE770X3.97-1983, with the following exceptions:

- Identifiers have unlimited length. See "Identifiers" on page 33.
- The at symbol (`@`) is not equivalent to the caret (`^`). See "The @ Operator" on page 101.
- Values that are assigned to pointers can be obtained in ways other than from the `New` procedure. See "Pointer Types" on page 53.
- A CodeWarrior Pascal string type variable of length *n* consists of a length byte followed by *n* characters, whereas an ANS string variable would consist of a length byte followed by *n*-1 characters. See "String Types" on page 51.
- In CodeWarrior Pascal, the type `TEXT` is distinct from the type `FILE OF CHAR`. The type `FILE OF CHAR` is a file whose records are of type `CHAR`, containing `CHAR` values that are not interpreted or converted in any way during I/O operations. See "Pascal File Variables" on page 182.
- The ANS Pascal comment delimiters `{}` and `(* *)` are used to allow comment nesting, so the Standard comment delimiters `{**)` and `(**}` are not supported. See "Comments" on page 38.
- Program parameters in the `PROGRAM` statement, as described by Jensen and Wirth and the ANS Standard, are ignored (but still required) by CodeWarrior Pascal.

Extensions to ANS Pascal

In addition to the requirements of the ANS standard, the CodeWarrior implementation of Pascal includes several extensions. Note that these extensions are active only if the compiler is set *not* to comply with the ANS standard. For information about how to set this and other compiler options, see the *Pascal Compilers Reference*.

The following list outlines the CodeWarrior Pascal extensions to the ANS standard:

- Constant expressions are allowed in declarations and array indexes. See "Arrays And String Indexes" on page 82.
- Declarations can be written in any order. See "Block Structure" on page 161 and "Unit Structure" on page 170.
- The `CYCLE` statement is supported. See "The CYCLE Statement" on page 122.
- The `LEAVE` statement is supported. See "The LEAVE Statement" on page 123.
- Ranges are allowed in `CASE` statement tags. See "The CASE Statement" on page 120.
- The `Exit` procedure is supported. See the "Exit and Halt" chapter of the *Pascal Library Reference* for details.
- The *short-circuit OR*(`|`) and the *short-circuit AND* (`&`) operators are supported. See "Boolean Operators" on page 96.
- Functions can return values of structured types. See "Function Declarations" on page 127.
- `UNIV` parameters are supported. See "UNIV Parameters" on page 142.
- Type coercion techniques are supported. See "Type Coercion" on page 69.
- There are built-in bit manipulation routines. See the "Bit Manipulation" chapter of the *Pascal Library Reference*.
- The exponentiation operator `**` is supported. See "Arithmetic Operators" on page 94.
- Additional primitive File and I/O functions are supported. See the "Input and Output" chapter of the *Pascal Library Reference* for details.
- In 68k Mac OS, the routines `Cosh`, `Sinh`, `Tanh`, and `Log10` are supported but may be implemented differently depending on the presence or absence of a floating point co-processor; see the "68k Notes" chapters of the *Pascal Compilers Reference* for details.
- Qualifier identification is supported (in other words, preceding an object's identifier with the name of the unit in which it is declared is allowed). See "Variable Accesses" on page 80. for details.

- CodeWarrior Pascal allows the propagation of identifiers from parent units. See "The USES Clause" on page 174.
- CodeWarrior Pascal has the `UNSIGNEDWORD` and `UNSIGNEDLONG` data types ("UNSIGNEDWORD" on page 48 and "UNSIGNEDLONG" on page 48).
- CodeWarrior Pascal supports C-style strings. See "C Strings" on page 53.
- CodeWarrior Pascal supports constant parameters. See "Constant Parameters" on page 137.
- CodeWarrior Pascal supports variable initialization. See "Variable Initialization" on page 78.
- CodeWarrior Pascal allows more than numeric characters for labels. See "The GOTO Statement" on page 121.
- CodeWarrior Pascal has Object Pascal extensions. See "Object Pascal Overview" on page 147.
- CodeWarrior Pascal supports schemata. See "Schemata" on page 64.
- CodeWarrior Pascal supports typeless parameter passing. See "Typeless Parameters" on page 138.

Implementation-Dependent Features

ANSI/IEEE770X3.97-1983 defines several requirements that are classified as *implementation-dependent*. The standard uses the term implementation-dependent to describe a feature which may vary from one compiler to the next, leaving the compiler free to choose the most reasonable implementation based on the processor it is compiling for.

The effect of using a feature of CodeWarrior Pascal that is classified as implementation-dependent is unspecified. Programs that use these features should not depend on any specific course being chosen, because the results may be unpredictable. This leaves CodeWarrior Pascal free to choose the course that is best at the time.

CodeWarrior Pascal and MPW Pascal

CodeWarrior Pascal contains a several exceptions and extensions to Apple's MPW Pascal. These differences are described in the following sections:

- Exceptions to MPW Pascal
- Extensions to MPW Pascal

Exceptions to MPW Pascal

The following features differ between CodeWarrior Pascal and MPW Pascal:

- Identifiers have unlimited length instead of a 63 character limit. See "Identifiers" on page 33.
- The predefined numeric constants `MaxCOMP`, `MinNormREAL`, `MinNormDOUBLE`, `MinNormEXTENDED`, and `ComSpecs` are not supported.
- Not all compiler directives described in MPW Pascal are supported by the CodeWarrior Pascal compiler. See the "Directives, Pragmas, and Predefined Symbols" chapter of the *Pascal Compilers Reference* for details about which compiler directives are and aren't supported.
- On the Macintosh, `INF` and `NAN` are variables defined in the Mac OS math library, they are not constants.
- CodeWarrior Pascal accepts but disregards variant record selectors in the `New` and `SizeOf` predefined routines. See the "Memory Management" chapter of the *Pascal Library Reference* for more information on these routines.

Extensions to MPW Pascal

The CodeWarrior implementation of Pascal includes the following extensions to MPW Pascal:

- The `Inc` and `Dec` predefined procedures are supported. See the "Ordinal Operations" chapter in the *Pascal Library Reference* for information.

- The `UpCase` predefined function. See the "Ordinal Operations" chapter in the *Pascal Library Reference* for information about predefined procedures and functions.
- Qualifier identification is supported (in other words, preceding an object's identifier with the name of the unit in which it is declared is allowed). See "Variable Accesses" on page 80. for details.
- CodeWarrior Pascal allows the propagation of identifiers from parent units. See "The USES Clause" on page 174.
- CodeWarrior Pascal has the `UNSIGNEDWORD` and `UNSIGNEDLONG` data types ("UNSIGNEDWORD" on page 48 and "UNSIGNEDLONG" on page 48).
- CodeWarrior Pascal supports C-style strings. See "C Strings" on page 53.
- CodeWarrior Pascal supports constant parameters. See "Constant Parameters" on page 137.
- CodeWarrior Pascal supports variable initialization. See "Variable Initialization" on page 78.
- CodeWarrior Pascal allows more than numeric characters for labels. See "The GOTO Statement" on page 121.
- CodeWarrior Pascal supports schemata. See "Schemata" on page 64.
- CodeWarrior Pascal supports typeless parameter passing. See "Typeless Parameters" on page 138.

CodeWarrior Pascal and Turbo Pascal

CodeWarrior Pascal has been extended for compatibility with code written in Turbo Pascal. Support for the Turbo Pascal extensions is optional. For details on how to set compiler options, see the "Pascal Settings" chapter of the *Pascal Compilers Reference*.

The only topic in this section is:

- Support for Turbo Pascal Language Extensions

Support for Turbo Pascal Language Extensions

CodeWarrior Pascal programs (in Turbo Pascal compatibility mode) can make use of the Turbo Pascal I/O (input/output) model and several other routines that provide compatibility with Turbo Pascal source code. Complete coverage of predefined routines can be found in the *Pascal Library Reference*. For details about the Pascal files and file variables, see "Files and I/O Overview" on page 181.

3

Symbols

This chapter explains the concept of a *symbol* and describes how symbols fit together to form Pascal source code. In addition, it details each of the different kinds of symbols supported by CodeWarrior Pascal.

Symbols Overview

Symbols are the smallest meaningful elements of a computer language. Every Pascal source text is comprised of a succession of symbols. Each symbol is written as a string of ASCII characters, and must adhere to the following constraints:

- Each symbol must be complete and unbroken; one symbol may not be inserted within another.
- Comments can be inserted anywhere, as long as they do not break up other symbols.

  The Pascal compiler ignores comments. Comments are discussed in "Comments" on page 38.
- Subject to certain exceptions, explained later in this chapter, delimiters must be used alternately with the other symbols.

  The compiler uses delimiters to determine where other symbols begin and end. For a description of delimiters, see "Delimiters" on page 38.

The character set used by CodeWarrior Pascal is eight-bit extended ASCII, with characters represented by numeric codes in the range 0..255. The compiler does not recognize the ASCII control codes (ASCII 0 through ASCII 31), except tab and carriage return. Otherwise, it processes the rest of the ASCII character set (32 to 127), as well as the characters represented by the values (128-255). On Mac OS, this range comprises the Apple Macintosh Extended Character set, but since there is no cross-platform standard, characters in this range may vary from one platform to another.

The rest of this chapter discusses the following topics:

- Special Symbols And Reserved Words
- Identifiers
- Numbers
- Labels
- Quoted String Constants
- Quoted Character Constants
- Delimiters
- Comments
- Compiler Directives

Special Symbols And Reserved Words

Special symbols and *reserved words* are symbols having fixed meanings. If you try to change their meanings or use them in ways other than their intended uses, CodeWarrior Pascal will issue an error. The following single characters are special symbols:

```
+  –  *  /  =  <  >  [  ]  .  ,  (  )  :  ;  ^
@  {  }  $  &  |
```

The following character pairs are special symbols:

```
<>   <=   >=   :=   ..   (*   *)   (.   .)   **
```

Some of the special symbols are also operators. Operators are introduced in "Expression Syntax" on page 91.

NOTE: The symbols (. and .) are equivalent to [and].

The reserved words in CodeWarrior Pascal are listed in Table 3.1.

Table 3.1 Reserved words

AND	DOWNTO	IF	OF	THEN
ARRAY	ELSE	IMPLEMENTATION	OR	TO

ASM	END	IN	OTHERWISE	TYPE
BEGIN	EXTERNAL	INLINE	PACKED	UNIT
C	FILE	INTERFACE	PROCEDURE	UNIV
CASE	FAR	LABEL	PROGRAM	UNTIL
CONST	FOR	LEAVE	RECORD	USES
CYCLE	FORWARD	MOD	REPEAT	VAR
DIV	FUNCTION	NIL	SET	WHILE
DO	GOTO	NOT	STRING	WITH
OBJECT	INHERITED	OVERRIDE		

These reserved words appear in uppercase letters throughout this document. However, CodeWarrior Pascal is not case sensitive–corresponding uppercase and lowercase letters are equivalent.

Identifiers

Identifiers are the names that denote constants, types, variables, procedures, functions, units and programs, and fields in records. Here are the rules for writing identifiers:

- An identifier can be of any length but only the first 255 characters are significant.
- They are not case sensitive; corresponding uppercase and lowercase letters are equivalent.
- They may contain only letters, digits, and underscore characters ("_", ASCII 95); in particular, they may not contain spaces.
- Every identifier must begin with a letter or an underscore.

In the language of our syntax diagrams:

<identifier> ::=

(<letter> | <underscore>) {<letter> | <underscore> | <digit>}

Here are some examples of identifiers:

```
Z      Knowledge      SUM      get_byte
An_identifier_can_be_as_long_as_you_want
```

Numbers

Within a CodeWarrior Pascal program, you can use ordinary decimal notation for numbers that are constants of the integer and real data types. You can also write hexadecimal integer constants using the `$` character as a prefix. You can use scientific notation (`E` or `e` followed by an exponent) for real types. The exact syntax for specifying numbers is described by the following syntax diagrams:

<unsigned number> ::=

<digit sequence> | (**$** <hex digit sequence>)

<real-type numbers> ::=

[<sign>] (**INF** | (**NaN** [**(** <digit sequence> **)**]) | (<digit sequence> [**.** [<digit sequence>]] [**E** [<sign>] <digit sequence>]))

<digit sequence> ::=

<digit> {<digit>}

<hex digit sequence> ::=

<hex digit> {<hex digit>}

<sign> ::=

+ | **-**

The letter `E` or `e` preceding the scale factor in an unsigned real means “times ten to the power of.”

These are examples of correct notation for numbers in CodeWarrior Pascal programs:

```
1      +100      –0.1      5E–3      87.35e+8
$A05D
```

Notice that `5E–3` means $5x10^{-3}$ and `87.35e+8` means $87.35x10^{8}$. You can omit the plus sign (+) before the exponent; `8E+7` and `8E7` are equivalent.

Numbers written with a decimal point or exponent are stored as type `REAL` (unless explicitly assigned to a variable of another of the real types: `SINGLE`, `LONGREAL`, `DOUBLE`). Other decimal numbers are stored as the smallest numerical type (`INTEGER` or `LONGINT`) needed to represent that value. Information about the range of values which can be represented by the `INTEGER` and `LONGINT` data types can be found under "Scalar Types" on page 44. A number which can be represented by neither the `INTEGER` nor the `LONGINT` datatype will cause an overflow error.

A hexadecimal constant is similarly stored as either an `INTEGER` or a `LONGINT`, depending on which is the smallest type that can represent the value. Each two digits of a hexadecimal constant require one byte to be represented. In this way, a hexadecimal number with one to four digits requires two bytes; one with five to eight digits requires four bytes (Information about the actual sizes of the `INTEGER` and `LONGINT` data types can be found in the platform "Notes" chapters of the *Pascal Compilers Reference*.) An integral hexadecimal value which can be represented by neither the `INTEGER` nor the `LONGINT` datatype will cause an overflow error. Leading zeros are counted in hexadecimal digit counts. The sign of the resulting value is implied by the hexadecimal notation.

Here are some examples of hexadecimal constants and their integer values:

```
$F=15
$FFFF=-1  (assuming 2-byte, two's-complement integers.)
$0FFFF=65535
$FFFFF=1048575
```

For information on integer and real types, refer to "Data Types Overview" on page 41.

Labels

A *label* is a sequence of characters, just like an identifier. Labels are used with the GOTO statement.

See also "The GOTO Statement" on page 121.

Quoted String Constants

A *quoted string constant* is a sequence of zero or more characters from the Apple Macintosh Extended ASCII character set.

NOTE: Characters outside of the standard ASCII character set range (0..127) will represent the Apple Macintosh Extended ASCII characters only if code is being generated for the Macintosh platform. Characters outside this range of values may be used for other platforms, but the characters represented can vary from one platform to the next.

Here are the rules for writing quoted string constants:

- Each constant must be written all on one line of the program source text.
- Each must be enclosed by single quotation marks (apostrophes).
- Blanks count as characters in quoted string constants.
- The maximum number of characters in one constant is 255.
- A quoted string constant with nothing between the single quotation marks denotes the null string.
- If you want the quoted string constant to contain a single quotation mark, you must write the single quotation mark twice.

The exact syntax for a quoted string constant looks like this:

<quoted string constant> ::=

' {<string character>} '

<string character> ::=

('') | (any character except ' or **Return**)

These are examples of quoted string constants:

```
'Baltic'   "NOVOGOROD" 'Don''t Panic!'
'A'          ';'          ''''          ''
```

The last is a *null string*. The next to last contains one single quotation mark.

All string values have a length attribute. In the case of a quoted string constant, the length is fixed; it is equal to the actual number of characters in the string value.

From CodeWarrior 8 forward, character strings enclosed in double quotation marks (") are treated as C-style strings. C strings have no initial length attribute but are terminated with the last byte set to zero.

See also "C Strings" on page 53.

Quoted Character Constants

Syntactically, a quoted character constant is simply a quoted string constant whose length is exactly one. `'A'` is an example of a quoted character constant:

<quoted character constant> ::=

' <string character> '

A quoted character constant is compatible with any `CHAR` type or `STRING` type; that is, it can be used either as a character value or as a string value.

Delimiters

Delimiters are symbols that separate other symbols in the source text so that the compiler can distinguish them as discrete objects. Blanks (spaces, tabs, carriage returns, and option-spaces) are the principal delimiters. In addition, all the special symbols listed earlier in this chapter serve as delimiters while performing their other functions. Hence the compiler can process the expression

```
2+seven=number_of_planets
```

even though it contains no spaces or tabs, because + and = are delimiters.

Comments and compiler directives also act as delimiters. For descriptions of comments and compiler directives, see "Comments" on page 38 and "Compiler Directives" on page 39.

Comments

The constructs

```
{ Any text not containing right-brace }
(* Any text not containing star-right-paren *)
// Any text to the end of one line
```

are called *comments*. They are ignored by the compiler.

A comment cannot be nested within another comment formed with the same kind of delimiters. However, a comment formed with `{...}` delimiters can be nested within a comment formed with `(*...*)` delimiters, and vice versa.

Comments formed with `//` are a C++ style comment that is allowed in the CodeWarrior implementation of Pascal.

NOTE: The use of nested comments is one of the differences between CodeWarrior Pascal and ANS Pascal. Nested comment structures allow you to "comment out" source text that contains only one style of comment delimiters–that is, render it invisible to the compiler. For more differences between CodeWarrior Pascal

and ANS Pascal, see "CodeWarrior Pascal and ANS Pascal" on page 23..

Compiler Directives

The CodeWarrior Pascal compiler supports a special set of commands, called *compiler directives*, which can be used to customize or control the compiler behavior from within Pascal source code. The syntax for compiler directives is that of a normal Pascal comment which contains a `$` character immediately after the `{` or `(*` that begins the comment.

For descriptions of the various compiler directives supported, see the "Directives, Pragmas, and Predefined Symbols" chapter in the *Pascal Compilers Reference*.

4

Data Types

This chapter describes how data types are used in a Pascal program as well as outlining the various predefined data types supported by CodeWarrior Pascal.

Data Types Overview

All data used in a Pascal program has an associated *data type*. A data type is fundamentally a mapping between a logical piece of data and its actual representation in machine memory. In Pascal, a logical piece of data is *declared* with a data type so the Pascal compiler will initially know how to build the data in actual machine memory, and then subsequently, how to operate on that segment of memory.

A data type must be specified whenever a variable is declared. And whenever a procedure or function is declared, a data type must be specified tor each parameter as well as the return value in the case of a function.

There are two kinds of data types, *user-defined* types and *built-in* types. Built-in types are types which are predefined by the compiler, while user-defined types are defined within a Pascal program or block. The first topic of this chapter explains the details of defining new types:

- Type Declarations

The chapter then continues by detailing each of the six categories of types built into CodeWarrior Pascal; some of these sections describe only predefined types, while others include relevant details about related user-defined types:

- Real Types
- Scalar Types
- String Types

- Pointer Types
- Procedure And Function Types
- Structured Types
- Opaque Types

The last few topics of the chapter focus on a few aspects of data type usage:

- Schemata
- Type Compatibility
- Type Coercion
- User-Defined Anonymous Types

Type Declarations

A *type declaration* associates an identifier with a type. Here is the syntax for a type declaration:

<type declaration> ::=

<identifier> **=** <type> **;**

When an identifier is declared as a certain type, the declaration is valid only within a certain region, depending on where the declaration appears. Type declarations can appear in either *blocks* or *units*. For information on declaring an identifier within a block, see Scope Rules"Scope Rules" on page 164. For information on declaring an identifier within a unit, see "Declaration Scope within Programs and Units" on page 173.

Here are some examples of identifiers being declared (Listing 4.1):

Listing 4.1 Example Type Declaration Part

```
TYPE count  = INTEGER;
    range  = INTEGER;
    color  = (red, yellow, green, blue);
    sex    = (male, female);
    year   = 1900..1999;
```

```
shape  = (triangle, rectangle, circle);
card   = ARRAY[1..80] OF CHAR;
str    = STRING[80];
polar  = RECORD r: REAL; theta: angle END;
person = ^personDetails;
personDetails = RECORD
 name, firstName: str;
 age: INTEGER;
 married: BOOLEAN;
 father, child, sibling: person;
 CASE s: sex OF
   male: (enlisted, bearded: BOOLEAN);
   female: (pregnant: BOOLEAN)
 END;
people  = FILE OF personDetails;
intfile = FILE OF INTEGER;
```

In Listing 4.1, `count`, `range`, and `INTEGER` denote identical types. The type `year` is compatible with, but not identical to, the types `range`, `count`, and `INTEGER`.

Real Types

Real types are data types which represent real numbers. There are four predefined real type in CodeWarrior Pascal:

- `REAL`
- `SINGLE`
- `DOUBLE`
- `LONGREAL`

The `REAL` type is identical to the `SINGLE` type. Both of these forms are supported because `REAL` is the ANS Pascal standard.

These are the possible values for real-type variables:

- Finite values (a subset of the mathematical real numbers). The value zero has a sign, like other numbers, which appears in textual output.

- Infinite values, `+INF` and `–INF`. These arise either as the result of an operation that overflows its intended storage type or as the result of dividing a finite value by zero.
- NaN (the word *NaN* stands for Not a Number). NaNs arise as the result of operations that have no meaningful numeric result. For example, the result of multiplying ∞ by zero is a NaN. In textual output a NaN appears as `NAN`, followed by a set of parentheses enclosing an integer that identifies the source of the NaN.

The ranges of values that can be represented by the various real data types are dependent on the compiler and compiler settings being used, but in general the `REAL` and `SINGLE` types are equivalent and can represent the same range of values, while the `LONGREAL` and `DOUBLE` types require more memory space and can represent a larger range of values. For more specific information, see the *Pascal Compilers Reference*.

Scalar Types

Scalar types are types which are based on the notion of a discrete set of ordered values, such as the set of whole numbers. All scalar types adhere to the following rules:

- Within a given scalar type, all possible values form an ordered set and each possible value is associated with an ordinality, which is an `INTEGER` or `LONGINT` value. Except for `INTEGER` and `LONGINT` values, the first value of the scalar type has ordinality 0, the next has ordinality 1, and so on for each value in that scalar type. The ordinality of an `INTEGER` or `LONGINT` value is the value itself; for example, the ordinality of –10 is –10. In any scalar type, each value except the first has a predecessor based on this ordering, and each value except the last has a successor based on this ordering.
- The standard function `Ord`, described in the "Ordinal Operations" chapter of the *Pascal Library Reference*, can be applied to any value of scalar type; it returns the ordinality of the value.
- The standard function `Pred`, described in the "Ordinal Operations" chapter of the *Pascal Library Reference*, can be applied

to any value of scalar type; it returns the predecessor of the value. For the first value in the scalar type, the result is unspecified.

- The standard function `Succ`, described in the "Ordinal Operations" chapter of the *Pascal Library Reference*, can be applied to any value of scalar type; it returns the successor of the value. For the last value in the scalar type, the result is unspecified.
- The predefined procedure `Dec`, described in the "Ordinal Operations" chapter of the *Pascal Library Reference*, can be applied to any value of scalar type; it decrements its argument to its predecessor. For the first value in the scalar type, the result is unspecified.
- The predefined procedure `Inc`, described in the "Ordinal Operations" chapter of the *Pascal Library Reference*, can be applied to any value of scalar type; it increments its argument to its successor. For the last value in the scalar type, the result is unspecified.

Scalar types have the following syntax:

<scalar type> ::=

<subrange type> | <enumerated type> | <ordinal type identifier>

CodeWarrior Pascal supports the following predefined scalar types:

- `INTEGER`
- `LONGINT`
- `UNSIGNEDBYTE`
- `UNSIGNEDWORD`
- `UNSIGNEDLONG`
- `BOOLEAN`
- `CHAR`
- `COMP`

and two classes of user-defined scalar types:

- Enumerated Types

- Subrange Types

These types are described in the following sections.

INTEGER

Values of type INTEGER are a subset of the whole numbers. As constants, these values can be denoted as described in "Numbers" on page 34. The range of the type INTEGER is the set of values

```
–MaxINT - 1, –MaxINT,...,
–1, 0, 1, ... MaxINT–1, MaxINT
```

The actual value of the constant MaxINT is dependent on the specific compiler and platform being used, and can be found in the platform "Notes" chapters of the *Pascal Compilers Reference*.

LONGINT

Values of type LONGINT are a subset of the whole numbers. As constants, these values can be denoted as described under "Numbers" on page 34. The range of the type LONGINT is the set of values

```
–MaxLONGINT–1, –MaxLONGINT, ...–1, 0, 1,
... MaxLONGINT–1, MaxLONGINT
```

The actual value of the constant MaxLONGINT is dependent on the specific compiler and platform being used, and can be found in the platform "Notes" chapters of the *Pascal Compilers Reference*.

In general, the LONGINT type can represent a larger range of numbers than is possible with the INTEGER type. Arithmetic performed on operands of type INTEGER and LONGINT adheres to the following set of rules:

- All integer constants in the range of type INTEGER are considered to be of type INTEGER. All integer constants in the range of type LONGINT, but not in the range of type INTEGER, are considered to be of type LONGINT.
- When both operands of an operator (or the single operand of a unary operator) are of type INTEGER, the result is of type INTEGER (truncated if it falls outside the range of values

which can be represented by that type). Similarly, if both operands are of type LONGINT, the result is of type LONGINT.

- When one operand is of type LONGINT and the other is of type INTEGER, the INTEGER operand is converted to LONGINT, the result is of type LONGINT. However, if this value is assigned to a variable of type INTEGER, it is truncated (see next rule).
- The expression on the right of an assignment statement is evaluated independently of the type of the variable on the left. For example, if variable `longVar` is declared as type LONGINT, the statement `longVar := MaxINT+MaxINT` will still cause INTEGER overflow, even if the sum is within the range of LONGINT numbers. If necessary, the result of the expression is truncated or extended to match the size of the variable on the left.

An important point to remember is that each operator is applied only to its two operands, so, at most, one of those operands is converted. If the expression contains other operands, those are not necessarily converted. For example, in the expression

```
oneInt+twoInt+threeInt+oneLongint
```

the value of `oneInt` is added to `twoInt` and the result is of type INTEGER, which is added to `threeInt` resulting in another INTEGER-typed value, which is converted to a LONGINT value and added to `oneLongint`. The result of the expression is a LONGINT.

The `Ord4` function described in the "Ordinal Operations" chapter of the *Pascal Library Reference* can be used to convert an INTEGER value to a LONGINT value.

TIP: Although it varies on a platform by platform basis, it is not uncommon for operations performed on LONGINT values to take one and a half times as long as the equivalent operation performed on INTEGER values, and even more than twice as long in the case of multiplication and division operations. Specific information about INTEGER and LONGINT implementations on your platform, along with efficiency and memory implications, can be found

in the platform "Notes" chapters of the Pascal Compilers Reference.

UNSIGNEDBYTE

Values of type UNSIGNEDBYTE are a subset of positive whole numbers. This type can be used in expressions with the UNSIGNEDWORD, UNSIGNEDLONG, INTEGER and LONGINT types, but no range checking is available.

The range of the type UNSIGNEDBYTE is the set of values 0 to +255. This data type represents 8-bit unsigned integers.

UNSIGNEDWORD

Values of type UNSIGNEDWORD are a subset of positive whole numbers. This type can be used in expressions with the UNSIGNEDLONG, INTEGER and LONGINT types, but no range checking is available.

The range of the type UNSIGNEDWORD is the set of values

```
0, 1, ..., 2 * MaxINT, (2 * MaxINT ) + 1
```

UNSIGNEDLONG

Values of type UNSIGNEDLONG are a subset of positive whole numbers. This type can be used in expressions with the UNSIGNEDWORD, INTEGER and LONGINT types, but no range checking is available.

The range of the type UNSIGNEDLONG is the set of values

```
0, 1, ..., 2 * MaxLONGINT, (2 * MaxLONGINT ) + 1
```

BOOLEAN

The values of the BOOLEAN type are truth values denoted by the predefined constant identifiers FALSE and TRUE.

Specifics of the implementation and storage requirements of the BOOLEAN type can be found in the *Pascal Compilers Reference*.

CHAR

A variable of type CHAR holds extended eight-bit ASCII values, represented by numeric codes in the range 0..255. The ordering of the CHAR values is defined by the ordering of these numeric codes. The function call

```
Ord(c)
```

where *c* is a CHAR value, returns the numeric code of *c*.

COMP

CodeWarrior Pascal allows you to define an integer larger than LONGINT with the type specifier COMP. The actual size and range of possible values for the COMP type are discussed in the *Pascal Compilers Reference*.

Enumerated Types

An enumerated type defines an ordered set of values by listing the identifiers that denote these values. The ordering of these values is determined by the sequence in which the identifiers are listed. Below is the syntax for an enumerated type:

<enumerated type> ::=

(<identifier list> **)**

<identifier list> ::=

<identifier> { **,** <identifier>}

Enumerated identifiers are constants for the block in which they are declared. The type of this constant is the enumerated type being declared. These values are constants of the enumerated type in the same way that the characters 'A', 'B', and 'C' are constants of type CHAR and the integers 1, 2, and 3 are constants of type INTEGER.

These are examples of enumerated types:

```
color = (red, yellow, green, blue);
suit = (club, diamond, heart, spade);
```

Given these declarations, yellow is a constant of type color, diamond is a constant of type suit, and so forth. When the `Ord` function is applied to a value of an enumerated type, it returns an `INTEGER` representing the ordering of the value with respect to the other values of the enumerated type. For example, given the declarations above, `Ord(red)` returns zero, `Ord(yellow)` returns one, and `Ord(blue)` returns three.

For information on memory requirements of enumerated types, see the platform "Notes" chapters of the *Pascal Compilers Reference*.

See also "Identifiers" on page 33.

Subrange Types

You define a subrange type by giving a range of values from some scalar type, called the *associated scalar type*. A subrange type provides for range checking of values within the associated scalar type. The syntax for a subrange type is shown in the following syntax diagram:

<subrange type> ::=

    <constant expression> `..` <constant expression>

Both constants must be of scalar type. The first constant expression in a subrange type declaration must be smaller than the second constant expression. Both must be of the same scalar type, or one must be of type `INTEGER` and the other of type `LONGINT`. If one is of type `INTEGER` and the other of type `LONGINT`, the associated scalar type is `LONGINT`.

NOTE: When using a constant expression in a type declaration that is declaring a subrange type, you cannot use a parenthesis as the first character after the equal sign. The compiler distinguishes subrange types from enumerated types by the first symbol after the equal sign: a left parenthesis in that position signifies an enumerated type. If a subrange specification needs parentheses, pre-

cede it with `0+`. This rule applies only within the type declaration part of a program.

Here are some examples of subrange types:

```
1..100
–10..+10
red..green
0+(const1–const2) DIV 2..const2 {see NOTE above}
```

A variable of a subrange type possesses all the properties of variables of the associated scalar type, with the restriction that its runtime value must be in the specified interval. In addition, the variable may require less memory space. See the platform "Notes" chapters in the *Pascal Compilers Reference* for details.

See also "Scalar Types" on page 44.

String Types

A string data type represents a sequence of characters with arbitrary length. CodeWarrior Pascal supports two kinds of strings:

- Pascal Strings
- C Strings

Pascal Strings

An example of a pascal string type declaration is

```
TYPE
  aString = STRING[15];
```

where 15 is the maximum size of the string. The size is the maximum on the length of any value of this type. The size attribute of a string type is determined when the string type is defined, and cannot change. It has a value in the range 1..255. A string type declared without a size attribute is treated as `STRING[255]`.

A string type has the following syntax:

<string type> ::=

(**STRING** [**[** <size attribute> **]**]) | (<string type identifier>)

<size attribute> ::=

<constant expression>

The length is the actual number of characters in the sequence at any time during program execution. The current value of the length attribute is returned by the standard function `Length`.

The ordering relationship between any two string values is determined by the ordering relationship of character values in corresponding positions in the two strings. A capital letter does not have the same ordering value as the corresponding lowercase letter; for example, *A* is valued lower than *a*.

See "Comparing Strings" on page 100.

Remember that the size of a string is the value of the size attribute assigned to the string type when it is declared, and the length of a string is the number of characters it holds at any point, regardless of its size attribute. A program can measure the actual length of a string by using the `Length` function described in the "String Manipulation" chapter of the *Pascal Library Reference*.

NOTE: With a string constant, the size attribute is equal to the length—that is, the number of characters actually in the string.

Although string types are simple types by definition, they have some characteristics of structured types. As explained under "Array Types" on page 56, individual characters in a string can be accessed as if they were components of an array. In addition, all string types are implicitly packed types and all restrictions on packed types apply to strings. A list of restrictions is given in "Structured Types" on page 55.

Operators applicable to strings are discussed in "String Operators" on page 98. Predeclared procedures and functions for manipulating

strings are described the "String Manipulation" chapter of the *Pascal Library Reference*.

A Pascal string is stored as a one-byte-length field followed by the characters in the string. You can therefore change the length of the string by changing its zeroth character. For example,

```
myStr[0] := Chr(Ord(myStr[0])+n);
```

changes the length of `myStr` by the value of `n`.

C Strings

A C string, denoted in CodeWarrior Pascal as a literal string surrounded by double quotation marks (`"`), doesn't have an initial length attribute. Instead, the end of a C string is marked by a zero after the last character in the string.

When assigned to an object of type `STRING`, a C string literal is converted into a Pascal string. When assigned to an object of type `PACKED ARRAY OF CHAR` a C string literal is copied without translation and is terminated with a zero character if there is enough room, based on the array's declared size.

See also "PACKED ARRAY OF CHAR" on page 58.

Pointer Types

You can use the pointer type to define a pointer variable—a variable that holds a memory address. When you declare a pointer variable, you must specify the data type of the memory area it points to, which is then called the *base type* of that pointer variable.

The syntax for a pointer type is illustrated in the following syntax diagram:

<pointer type> ::=

(<pointer symbol> <base type>) | <pointer type identifier>

<base type> ::=

<type identifier>

The base type may be an identifier that has not yet been declared. In this case, it must be declared somewhere in the same type declaration part as the pointer type.

Aside from an address, any pointer variable can also hold the value `NIL`. You cannot assign the value of a pointer variable of one type to a pointer variable of another type, even if the first pointer variable has the value `NIL`.

You can create a pointer in three ways:

- By using the `New` procedure described in the "Memory Management" chapter of the *Pascal Library Reference*. This allocates a new memory area in the application heap for a *dynamic variable* and points the pointer variable to it. The size of the area is determined by the base type of the pointer variable, including optionally specified variants; see the description of `New` in the same chapter of the *Pascal Library Reference*. A dynamic variable is a variable that has no identifier of its own; the only way to access one is through a pointer.
- By using the `@` operator. The `@` operator points to the memory area occupied by any existing variable. The `@` operator pointer function creates a pointer that is compatible with all other pointer types.
- By using the `Pointer` function described in the "Ordinal Operations" chapter of the *Pascal Library Reference*. This allows any pointer to be coerced to any other pointer type.

See also "The @ Operator" on page 101.

Procedure And Function Types

CodeWarrior Pascal allows procedure and function types. Procedure and function types are also know as procedural types. You can use the procedure or function type to define a procedure or function variable or a procedural or functional parameter.

A procedure assigned to a procedure-type variable or procedural parameter must have formal parameters that are parameter list

compatible. Likewise, a function assigned to a function-type variable or functional parameter must have formal parameters and a return value that are parameter list compatible.

Here is the actual syntax for procedure and function types:

<procedure type> ::=

PROCEDURE <formal parameter list>

<function type> ::=

FUNCTION <formal parameter list> **:** <type identifier>

See also "Procedural Parameters" on page 138, "Functional Parameters" on page 141, and "Parameter List Compatibility" on page 143.

Structured Types

A *structured type* is a data type that stores more than one value. Each structured type is characterized by its structuring method and by the type or types of its components. If the component type is itself structured, the resulting structured type exhibits more than one level of structuring. There is no specified limit on the number of levels of structuring a data type can have.

The topics in this section are:

- About Structured Types
- Array Types
- `PACKED ARRAY OF CHAR`
- Record Types
- Set Types
- File Types
- Object Types

About Structured Types

The syntax diagram below describes the actual syntax for a structured type:

<structured type> ::=

([**PACKED**] (<array type> | <set type> | <file type> | <record type>)) | <stuctured type identifier> | <object type identifier>

The use of the word PACKED in the declaration of a structured type indicates to the compiler that data storage should be economized, even if this causes less efficient access to a component of a variable of this type. Although you can use the word PACKED when declaring any structured type, PACKED only affects the storage of record and array types.

The word PACKED only affects the representation of one level of the structured type in which it occurs. If a component is itself structured, the component's representation is packed only if the word PACKED also occurs in the declaration of its type.

NOTE: The implementation of packing is complex; details of memory allocation to components of a packed variable as well as restrictions on using these components are outlined in the platform "Notes" chapters of the Pascal Compilers Reference.

Array Types

An array type consists of a fixed number of components that are all of one type, called the *component type*. The number of elements is determined by one or more *index types*, one for each dimension of the array. There is no specified limit on the number of dimensions. In each dimension, the array can be indexed by every possible value of the corresponding index type, so the number of elements is the product of the number of values in each of the index types.

The syntax for an array type is described by the following pair of syntax diagrams:

<array type> ::=

ARRAY [<index type> { **,** <index type> } **] OF** <type>

<index type> ::=

<ordinal type identifier>

The type following the word `OF` is the component type of the array and can be an existing type identifier or a new type.

NOTE: The index type cannot be `LONGINT` or a subrange of `LONGINT`.

Here are some examples of array types (Listing 4.2):

Listing 4.2 Example Array Types

```
ARRAY[1..100] OF REAL
ARRAY[boolean] OF Color
ARRAY[1..Pagesize—1] OF CHAR
```

If the component type of an array type is also an array type, the result can be regarded either as an array of arrays or as a single multidimensional array. For example,

```
ARRAY[BOOLEAN] OF ARRAY[1..10] OF ARRAY[size] OF
REAL
```

is equivalent to

```
ARRAY[BOOLEAN, 1..10, size] OF REAL
```

Likewise,

```
PACKED ARRAY[1..10] OF PACKED ARRAY[1..8] OF
BOOLEAN
```

is equivalent to

```
PACKED ARRAY[1..10,1..8] OF BOOLEAN
```

"Equivalent" means that CodeWarrior Pascal performs the same actions with the two constructions.

A component of an array can be accessed by following the array's identifier with one or more indexes in brackets, separated by commas. For example, the two expressions

```
myArray[5,4]
myArray[5][4]
```

both access the fourth element in the fifth subarray of the array `myArray`. For further information, see "Arrays And String Indexes" on page 82.

PACKED ARRAY OF CHAR

Like the `STRING` type the `PACKED ARRAY OF CHAR` can be used to hold a character string. Unlike the `STRING` type, `PACKED ARRAY OF CHAR` doesn't have an initial length byte. Unless dealing with C strings, CodeWarrior Pascal considers the length of a string in a `PACKED ARRAY OF CHAR` to be the size of the array.

See also "String Types" on page 51, "Pascal Strings" on page 51, "C Strings" on page 53, and "Array Types" on page 56.

Record Types

A record type consists of a fixed number of components called *fields*, which can be of different types. For each component, the record type declaration specifies the type of the field and an identifier that names the field (the exact syntax will be described shortly.)

Record types consist of *fixed parts* and *variant parts*, as illustrated in the following syntax diagrams:

<record type> ::=

RECORD [<field list>] **END**

<field list> ::=

(<variant part> | (<fixed part> [**;** <variant part>])) [**;**]

The fixed part of a record type specifies a list of "fixed" fields, giving an identifier and a type for each field. Each fixed field contains data that is always accessed in the same way. Here is the syntax for the fixed part:

<fixed part> ::=

<field declaration> { **;** <field declaration> }

<field declaration> ::=

<identifier list> **:** <type>

Here is an example of a record type (Listing 4.3):

Listing 4.3 Example Record Type

```
RECORD
  year : INTEGER;
  month:1..12;
  day:1..31
END;
```

A variant part allocates memory space with more than one list of fields, thus permitting the data in this space to be accessed in more than one way. Each list of fields is called a *variant*. The variants overlay each other—that is, they occupy the same space in memory.

The syntax for the variant part looks as follows:

<variant part> ::=

CASE [<identifier> **:**] <tag field type> **OF** <variant> { **;** <variant> }

<tag field type> ::=

<ordinal type identifier>

<variant> ::=

<constant> { **,** <constant> } **:** **(** [<field list>] **)**

The variant part allows for an optional identifier, called the *tag field identifier*. If a tag field identifier is present, it is automatically declared as the identifier of an additional fixed field of the record, called the *tag field*. The value of the tag field may be used by the program to indicate which variant should be used at a given time. If there is no tag field, the program must select a variant on some other criterion.

NOTE: The type LONGINT cannot be used as a tag type.

Each variant is identified by one or more constants. All the constants must be distinct and must be of a scalar type that is the same as or compatible with the tag type. The constants that introduce a variant are not used for referring to fields of the variant; the actual field identifiers are used. However, these constants can be used as optional arguments with the `New` procedure described in the "Memory Management" chapter of the *Pascal Library Reference*.

Variant fields are accessed in exactly the same way as fixed fields.

Examples of record types with variants are (Listing 4.4):

Listing 4.4 Example Record Types With Variants

```
RECORD
  name, firstName: STRING[80];
  age: 0..99;
  CASE married: BOOLEAN OF
    true:  (spousesName: STRING[80]);
    false: ()
END;

RECORD
  x, y: REAL;
  area: REAL;
  CASE s: shape OF
    triangle:  (side: REAL;
        inclination, angle1, angle2: angle);
    rectangle: (side1, side2: REAL;
```

```
        skew, angle3: angle);
    circle:    (diameter: REAL)
END;
```

Set Types

A set type defines a group of values, each of which has the same scalar type, called the set's *base type*. Each possible value of a set type is some subset of the possible values of the base type. Set types are defined with the following syntax:

<set type> ::=

SET OF <scalar type>

When you create a variable of a set type, that variable can hold none, one, several, or all of the values of the set.

The set operators and the way in which set values are denoted in Pascal are discussed in "Set Operators" on page 98.

The empty set `[]` is a possible value of every set type.

NOTE: The compiler may impose limitations as to how many elements a set can contain and which types can be used as base types. See the platform "Notes" chapters in the Pascal Compilers Reference for details.

The actual implementation of set types as well as the associated performance implications are covered in the platform "Notes" chapters of the *Pascal Compilers Reference*.

Here are some examples of set types (Listing 4.5):

Listing 4.5 Example Set Types

```
SET OF char
SET OF (black, brown, red, yellow, white)
SET OF 1..10
```

```
names = (Eliot, Pound, Yeats)  {a new scalar type}
poets = SET OF names  {set type using the new type}
```

File Types

A file type is a structured type consisting of a sequence of components that are all of one type, the *component type*. The component type may be any type except a file type or any type containing a file type. The syntax for a file type looks as follows:

<file type> ::=

TEXT | (**FILE** [**OF** <type>])

The component data is not in program-addressable memory but is accessed by means of a peripheral device. The number of components (the length of the file) is not fixed by the file type declaration.

In CodeWarrior Pascal, files can be passed to procedures and functions as variable parameters.

Below, we discuss two different kinds of file types:

- `FILE`
- `TEXT`

For more information on manipulating files, see "Files and I/O Overview" on page 181.

FILE

The type `FILE` (without the `OF TYPE` construct) represents an *untyped file*, meaning there is no way to reference the data by array-style indices. See the "Input and Output" chapter of the *Pascal Library Reference* for details.

NOTE: Although the symbol `FILE` can be used as a type identifier, it cannot be redeclared because it is a reserved word.

TEXT

The predefined file type TEXT denotes a file of characters organized into lines. Files of type TEXT are supported by the specialized I/O procedures discussed in the "Input and Output" chapter of the *Pascal Library Reference*.

Object Types

An object type defines a structure for an object. An object type can have fields, like a record. The discussion about record type fields ("Record Types" on page 58) also applies to object type fields, except that an object type cannot have a variant part. In addition to fields, an object type can have associated procedures and functions, called methods. The next few syntax diagrams describe the exact syntax of an object type:

<object type> ::=

OBJECT [**(** <object type identifier> **)**] { <object data heading> | <method heading> } **END**

<object type identifier> ::=

<identifier>

<object data heading> ::=

<object data field scope> <field declaration>**;**

{<object data field scope> <field declaration> **;**}

<object data field scope> ::=

PROTECTED | **PRIVATE** | **PUBLIC**

<method heading> ::=

(<function heading> | <procedure heading>) [**; OVERRIDE**] { **;** <method heading> [**; OVERRIDE**] } [**;**]

The method heading has the syntax of a procedure or function heading, as introduced in Chapter 8, "Procedures & Functions."

If you include the optional object type identifier and period, it must be the object type identifier you are defining. If the method has a formal parameter list, that list must be given with the heading; similarly, if the method is a function, the type of the return value must be given with the heading.

For more information about object types, see Chapter 9, "Object Pascal."

Opaque Types

An anonymous type may be specified in the interface part of a unit while its actual type declaration appears in the implementation part. The corresponding type declaration in the implementation needs to be a pointer to some type or an object. The content of the opaque type is unavailable to any unit except to the defining implementation.

<opaque type> ::=

    <opaque type identifier>

<opaque type identifier> :==

    <identifier>

Schemata

A schema determines a collection of similar types. A schema makes it possible to set a variable or parameter's type and size both at compile time and runtime.

The topics in this section are:

- Schemata Overview
- Schemata Limitations

Schemata Overview

Listing 4.6 shows an example of using a schema to provide a variable-sized array. In Listing 4.6, a schema, `Vector`, is defined. When

passed a value, represented by its formal discriminant, `len`, the `Vector` schema represents an array of `len` elements of `REAL` values.

Listing 4.6 Example of a schema

```
CONST
   MaxVector = 100;

TYPE
   Vector(len : MaxVector) = ARRAY [1 .. len] OF REAL;

FUNCTION GetVectorSize : MaxVector;
VAR
   s : MaxVector;
BEGIN
   Write('How big?');
   ReadLn(s);
   GetVectorSize := s;
END;

VAR
   v1 : Vector(10); (* Size is set at compile time *)
   v2 : Vector(GetVectorSize); (* Size is set at runtime *)
   vPtr : ^Vector;
BEGIN
   New(vPtr, 25);
   (* ... *)
END.
```

The size and type of schemata are set at compile time and runtime by declaring variables globally or in a routine. Schemata may also set an item's the type and size of at runtime using the `New` runtime routine.

With a schema, the allocation procedure `New` may dynamically select the type (and thus the size) of the variable it allocates.

A schematic formal-parameter adjusts to the bounds of its actual-parameters. The declaration of a local variable may dynamically select the type (and thus the size) of the variable.

Formal schema discriminants can be used as variant selectors in record definitions. Listing 4.7 shows an example of a schema, `Color_Map`, that uses its formal discriminant, `map`, to set a record's definition.

Listing 4.7 Example of a schema for a record definition

```
TYPE
   Color = (red,yellow);
   Color_Map(map : Color) = record
      case formal_discriminant of
         red: (red_field : integer);
         yellow : (yellow_field : integer);
   end;
```

Schemata Limitations

CodeWarrior Pascal imposes these limitations to using schemata:

- two variables are not assignment-compatible unless they are declared in the same variable declaration.
- an item of a record data type, in which the record definition contains a schema field at any level, may not be assigned to another item of a record data type
- functions may not return a value of a schema type or a record containing a schema field
- object data members may not be of a schema type or a record containing a schema field
- array elements may not be of a schema type
- variable initialization is not permitted with schemata

See also "Type Compatibility" on page 66.

Type Compatibility

The Pascal compiler uses data types to check that variables, constants, and function call results are compatible whenever they are combined in the same expression. If you try to use incompatible

types, the compiler will issue an error. More information about compiler errors can be found in the "Pascal Settings" chapter of the *Pascal Language Reference*.

There are three levels of type compatibility in Pascal:

- Same Type Compatibility
- Simple Compatibility
- Assignment Compatibility

Same Type Compatibility

Types have *same-type compatibility* when they are declared using the same type identifier or when their definitions can be traced back to the same type identifier. For the rules under which user-defined anonymous types are the same, see "User-Defined Anonymous Types" on page 70.

The same types are required only:

- Between actual and formal variable parameters
- Between actual and formal result types of functional parameters
- Between actual and formal value and variable parameters within parameter lists of procedural or functional parameters
- When a one-dimensional `PACKED ARRAY OF CHAR` is being compared with another via a relational operator

Parameters are discussed in "Parameters" on page 133.

Whenever two types are "same type" compatible, they are also compatible in all situations where "simple compatibility" and "assignment compatibility" are required.

Simple Compatibility

Types with *simple compatibility* are required in many contexts where two or more entities are used together, such as in expressions, in relational operations, and with `FOR` statement control variables and their initial, final, and step values. Other specific instances where

type simple compatibility is required are noted elsewhere in this manual.

Two types exhibit simple compatibility if any of the following are true:

- They are the same type.
- One is a subrange of the other.
- Both are subranges of the same type.
- Both are string types (the lengths and sizes may differ).
- Both are set types, and their base types are compatible.
- Both are of type `PACKED ARRAY OF CHAR` and have the same number of components.

Assignment Compatibility

Assignment compatibility is required whenever a value is assigned to something, either explicitly (as in an assignment statement) or implicitly (as in passing value parameters).

A type `T2` is assignment compatible with another type `T1` in the expression `T1 := T2` if any of the following are true:

- `T1` and `T2` are identical types, and neither is a file type or a structured type that contains a file type component or schema at any level of structuring.
- `T1` is a real type (`REAL`, `SINGLE`, `LONGREAL`, `DOUBLE`, or a type derived from these types), and `T2` is type `INTEGER`.
- `T1` and `T2` are compatible scalar types, and the value of `T2` is within the range of possible values of `T1`. See also "Scalar Types" on page 44.
- `T1` and `T2` are compatible set types, and all the members of `T2` are within the range of possible values of the base type of `T1`.
- `T1` and `T2` are `STRING` types, and the current length of `T2` is equal to or less than the size attribute of `T1`.
- `T1` and `T2` are both type `PACKED ARRAY OF CHAR`.

- `T1` is a string type with size greater than zero or a `CHAR` type, and `T2` is a quoted character constant.
- `T1` is type `PACKED ARRAY[1..`*n*`] OF CHAR`, or actually has *n* elements, and `T2` is a string constant containing exactly *n* characters. This is not true, however, if *n* =1, because a string constant of length 1 is a quoted character constant.

Whenever assignment compatibility is required and none of the above is true, either a compiler error or a runtime error occurs.

Type Coercion

A value or variable access of one type can be changed into a value of another type with the syntax

```
typeID(x)
```

Using this construction, *x* can be a variable identifier, a variable identifier plus one or more qualifiers (array index, field designator, file buffer symbol, or pointer symbol), or an expression. You can use this syntax on the left or right side of an assignment statement. The term *typeID* stands for any type identifier.

NOTE: The Pascal compiler imposes certain restrictions on coercing variables, pertaining specifically to the storage sizes of the data types involved. For a treatment of such restrictions, see the platform "Notes" chapters of the Pascal Compilers Reference.

Here are some examples of type coercion (Listing 4.8):

Listing 4.8 Examples of Type Coercion

```
TYPE
  ARecord = RECORD
      x, y: INTEGER
    END;
VAR
  recordVar: ARecord;
  LongVar:  LONGINT;
```

```
  IntVar: INTEGER;
---
recordVar := ARecord(LongVar);
LONGINT(recordVar) := 34 + 65536*180;
IntVar  := INTEGER(LongVar);
```

The last line shows a conversion from a LONGINT to an INTEGER. Notice that this *is* allowed, even though the LONGINT value may be outside of the range of values that can be represented by an INTEGER. For information on how the compiler handles this situation, see the platform "Notes" chapters of the *Pascal Compilers Reference*.

CodeWarrior Pascal does not support type coercion of set variables.

User-Defined Anonymous Types

You can give a type to a variable without defining a type identifier. In that case, the type declaration appears on the right side of a variable declaration, in place of a type identifier. This is called a *user-defined anonymous type*. You can create the same structures with these types that you can with identified types, except that you cannot use them in the following contexts:

- Formal parameter lists
- Definitions of functional return values

Here are some examples of user-defined anonymous type declarations, appearing on the right sides of variable declarations (Listing 4.9):

Listing 4.9 User-defined Anonymous Type Declaration Example

```
VAR   var1: (red, yellow, green, blue);
    circus_name: String Types[80];
    var3: RECORD
      name, firstName: str;
      age: INTEGER;
      married: BOOLEAN;
      father, child, sibling, dog_boy: person;
```

```
CASE s: sex OF
   gentleman: (bearded, enlisted: BOOLEAN);
   lady:   (bearded, pregnant: BOOLEAN)
END;
```

User-defined anonymous types that are separately declared are never the same types. For example, the declarations

```
var1: RECORD
    a, b: REAL
  END;
var2: RECORD
    a, b: REAL
  END;
```

do not give `var1` and `var2` the same type, even though they appear to be exactly the same. However, two variables declared in the same user-defined anonymous type declaration, as in

```
var1, var2: RECORD
      a, b: REAL
    END;
```

are of the same type.

You can coerce two separately declared user-defined anonymous types into being treated as identical by using the type coercion technique described earlier in "Type Coercion" on page 69.

5

Constants and Variables

This chapter describes how constants and variables are used to refer to data in CodeWarrior Pascal programs.

Constants and Variables Overview

Data processed by a Pascal program can be stored in *constants* and *variables*. As indicated by their respective namings, a variable refers to data which may be change throughout the life of the program, while a constant refers to data which can not be changed.

The purpose of this chapter is to detail what kinds of constants and variables are supported by CodeWarrior Pascal, as well as to describe how they are created and used.

The topics discussed in the rest of this chapter are:

- Constants
- Variables

Constants

Constants are used in Pascal programs to store and keep track of data values that remain *constant* during the entire time they are in use.

The rest of the section covers the following topics about constants:

- Constant Declarations
- Predefined Constants

Constant Declarations

A constant declaration defines an identifier that denotes a constant within the program, block, or unit that contains the declaration. The scope of a constant identifier does not include its own declaration.

For more details on declaring constants within programs and units, see "Programs and Units Overview" on page 169. Declaring constants within blocks is discussed in "Block Structure" on page 161. Both of these sections contain more information about the *scope* of constant declarations.

The syntax of a constant declaration is as follows:

<constant declaration> ::=

<constant identifier> ((**=** <constant>) | (**:** <type> **=** (<constant> | <variable initialization>))) **;**

<constant identifier> ::=

<identifier>

<constant> ::=

<constant expression>

<variable initialization> ::=

((<identifier> **:** <variable initialization> | <constant>) [**,** <variable initialization>] **)**

For information on using constant expressions, and expressions in general, see "Expressions Overview" on page 87.

NOTE: In the constant declaration syntax above, constructions of the "<constant identifier> **:** <type> = <variable initialization>" variant actually represent variables as opposed to constants. This, strangely, is how a variable definition with initialization is expressed in CodeWarrior Pascal. See "Variable Initialization" on page 78for details.

Predefined Constants

There are several constants which can be used anywhere in a CodeWarrior Pascal program without being declared. These constants are referred to as *predefined constants*; their values are defined by the compiler itself.

The rest of this section lists the CodeWarrior Pascal predefined constants:

`CompDate`

`CompDate` is a predefined `STRING` value. It contains the date that it was compiled. At compile time, CodeWarrior Pascal replaces `CompDate` with the current date.

`CompTime`

`CompTime` is a predefined `STRING` value. It contains the time of day that it was compiled. At compile time, CodeWarrior Pascal replaces `CompTime` with the current time.

`MaxINT`

The predefined constant `MaxINT` is of type `INTEGER`. Its value is the greatest integer which can be represented by the `INTEGER` data type.

For information on the `INTEGER` data type, see "INTEGER" on page 46 as well as the "Notes" chapters of the *Pascal Compilers Reference* for specific details about the implementation of this data type and the actual value of `MaxINT` under different target platforms.

`MaxLONGINT`

The predefined constant `MaxLONGINT` is of type `LONGINT`. Its value is the greatest integer which can be represented by the `LONGINT` data type.

For information on the `LONGINT` data type, see "LONGINT" on page 46 as well as the "Notes" chapters of the *Pascal Compilers Refer-*

ence for specific details about the implementation this data type and the actual value of `MaxLONGINT`.

`NIL`

Conceptually, `NIL` is a predefined pointer type value that does not point to anything. You can assign `NIL` to any pointer variable, regardless of type.

One common usage is to assign `NIL` to a pointer variable rather than leaving it with an undefined value, primarily because you can test for `NIL`.

Variables

A variable refers to data with a value that can be set at any time during program execution. This section describes how variables are created and used, including the various ways of accessing the data within a given kind of variable. The following topics are covered:

- Variable Declarations
- Variable Initialization
- Variable Accesses
- Qualifiers to Variable Accesses

Variable Declarations

A variable declaration, much like a constant declaration, gives the variable identifier meaning within the program, block, or unit within which it is declared. Unlike a constant, a newly declared variable does not necessarily have a value initially; declaring a variable simply sets up space for it and sets up the variable identifier to refer to that data within its valid scope. The result of using a variable before a value is assigned to it is undefined (CodeWarrior Pascal does provide away of initializing variables at the point of declaration; see "Variable Initialization" on page 78 for details.)

For more details on how to declare variables within programs and units, see "Programs and Units Overview" on page 169. Informa-

tion on declaring variables within blocks can be found in "Block Structure" on page 161. Both of these sections contain more information about the *scope* of variable declarations.

A variable declaration consists of a list of identifiers denoting new variables, followed by their type and optionally a directive (`EXTERNAL` or FAR, to be discussed shortly.) For a discussion of the significance of the type associated with a declared variable, see "Data Types Overview" on page 41.

Here is the syntax for a variable declaration:

<variable declaration> ::=

<identifier list> **:** <type> **;** [((**EXTERNAL**) | (**FAR**)) **;**]

The syntax for an identifier list is given under "Enumerated Types" on page 49.

NOTE: The `FAR` directive is only meaningful when targeting the 68k Macintosh platform. For details on far data addressing, see the chapter "68k Notes" in the Pascal Compilers Reference.

The type given for a variable can be a type identifier declared in a preceding type declaration part (which can be in the same block or an enclosing block, or in a unit) or a new type definition (a user-defined anonymous type). For information about anonymous types, see "User-Defined Anonymous Types" on page 70.

The values of all variables are undefined at the start of each activation of a block. The main program block is activated when the program is run. A procedure or function block is activated each time the procedure or function is called.

`EXTERNAL`

Using the `EXTERNAL` directive in a variable declaration signifies that the variable appears in a separate unit or library in the project and will be resolved at link time. Use the `EXTERNAL` directive to refer-

ence variables that appear in assembly language or C/C++ object code.

NOTE: Object Pascal objects are not directly compatible with C++ objects. For information on using C++ code with CodeWarrior Pascal, see the "Pascal Notes" chapter of the Pascal Compilers Reference.

Here are some examples of variable declarations (Listing 5.1).

Listing 5.1 Variable Declaration Examples

```
x, y, z: REAL;
i, j: INTEGER;
k: 0..9;
p, q, r: BOOLEAN;
operator: (plus, minus, times);
a: ARRAY[0..63] OF REAL;
c: color;
f: FILE OF CHAR;
hue1, hue2: SET OF color;
p1, p2: person;
m, m1, m2: ARRAY[1..10, 1..10] OF REAL; FAR;
coord: polar; EXTERNAL;
pooltape: ARRAY[1..4] OF tape;
```

Variable Initialization

In CodeWarrior Pascal, variables can be initialized at the point of declaration. Variables defined in this way are not declared in the variable declaration part of the block; instead, they are defined along with constants in the constant declaration part. The syntax for declaring a variable in this was is in "Constant Declarations" on page 74.

Variables declared with initial values are initialized only once and keep their values even after execution leaves the block they are defined in (this does not mean to imply that they are valid outside

their normal scope. Their usage is valid within the same scope as other variables.)

Pointer variables can be initialized to either NIL or an object that is not a local variable or parameter.

Fields in records must be initialized by specifying the field named, followed by a colon (:), follow by the value of the field.

Listing 5.2 Examples of variable initialization

```
PROCEDURE LeftProc;
BEGIN
(* ... *)
END;

PROCEDURE RightProc;
BEGIN
(* ... *)
END;

TYPE
  arrType = ARRAY [0 .. 2] OF CHAR;
  recType = RECORD
              a : ^recType;
              b : arrType;
              c : INTEGER;
            END;
  TProc   = PROCEDURE;
  TreePtr = ^ Tree;
  Tree    = RECORD
              left,
              right : TreePtr;
              proc: TProc;
            END;

CONST
  (* Constant definitions *)
  x = 'this is a constant string';
```

```
  (* Variable initializations *)
  xs : STRING = 'this is a variable';
  xi: INTEGER = 3;
  xa: arrType = ('a', 'b', 'c');
  xr: recType = (a : NIL, b : ('a', 'b', 'c'), c : 3);
  leftNode  : Tree =
    ( left : NIL, right : NIL, proc : LeftProc);
  rightNode : Tree =
    ( left : NIL, right : NIL, proc : RightProc);
  rootNode  : Tree =
    ( left : @leftNode, right : @rightNode, proc : NIL);
(* ... *)
```

Variable Accesses

When a variable's identifier is used in a program, it is called an *access* of that variable. You use a variable access to do any of the following:

- Obtain the value of a variable
- Assign a value to a variable
- Pass the value of a variable to a procedure or function

The object accessed may be any one or a combination of the following:

- A simple variable
- A pointer variable
- The collection represented by a variable of structured type
- A part of a structured type
- The identified variable of a pointer
- A variable reached through a function call

The syntax for accessing variables is as follows (qualifiers used with variable accesses are described in the section that follows):

<variable access> ::=

(<variable identifier> | <function call>) {<qualifier>}

<variable identifier> ::=

<identifier>

Qualifiers to Variable Accesses

In a variable access, the variable identifier or function call can be followed by one or more *qualifiers*. Each qualifier modifies the meaning of the variable access. This syntax diagram illustrates the different possible qualifiers:

<qualifier> ::=

<index> | <field designator> | <file buffer symbol> | <pointer symbol>

As indicated in the diagram for variable accesses, there can be none, one, or more than one qualifier following a variable identifier or function call, depending on the levels of structure in the variable and on which particular level you want to access.

For example, an array identifier with no qualifier is a reference to the entire array:

```
xResults
```

If the array identifier is followed by an index, this denotes a specific component of the array, which may be a simple or structured variable:

```
xResults[current+1]
```

If the array component is a record, the index may be followed by a field designator; in this case, the variable access denotes a specific field within a specific array component:

```
xResults[current+1].link
```

If the field is a pointer, the field designator may be followed by the pointer symbol to denote the variable pointed to by the pointer:

```
xResults[current+1].link^
```

If the variable identified by the pointer is an array, another index can be added to denote a component of this array, and so forth:

```
xResults[current+1].link^[i]
```

The remainder of this section details the following variable access qualifiers:

- Arrays And String Indexes
- Records and Field Designators
- Pointers and Their Identified Variables
- File Window Variables

Arrays And String Indexes

A specific component of an array variable is denoted by a variable access that refers to the array variable, followed by an index that specifies the component. The index part looks as follows:

<index> ::=

[<expression> { **,** <expression> } **]**

A specific character within a string variable or constant is denoted by a variable access, quoted string constant, or string constant identifier, followed by an index that specifies the character position.

The index of a string always ranges from 0 to a maximum of 255, depending on the length of the string. The index of an array depends on the ordinal type or types defined as the array's index type or types.

Each expression in the index selects a component in the corresponding dimension of the array. The number of expressions must not exceed the number of index types in the array declaration, and the type of each expression must be assignment compatible with the corresponding index type. For information on *assignment compatibility*, see "Type Compatibility" on page 66.

In indexing a multidimensional array, you can use either multiple indexes or multiple expressions within an index. The two forms are equivalent. For example,

```
m[i][j]
```

is equivalent to

```
m[i,j]
```

A string value can be indexed by only one index expression, whose value must be in the range 1..*n*, where *n* is the current length of the string value. The effect is to access one character of the string value.

NOTE: In general, you cannot assign a value to an individual character position in a string unless a character previously occupied that position. This is implied by the rule above, that an index into a string array can only have values between 1..n, where n is the current length of the string.

When a string value is manipulated by assigning values to individual character positions, the dynamic length of the string is not maintained. For example, suppose that `strval` is declared as follows:

```
strval: STRING[10];
```

Pascal allocates one byte for a number that represents the current length of the string, followed by space for ten `char` values. (The dynamic length is the number of `char` values in the string at any given time.) Initially, all of this space contains unspecified values. The assignment

```
strval[1] := 'F'
```

may or may not work, depending on what the unspecified length happens to be. If this assignment works, it stores the `char` value `F` in character position `1`, but the length of `strval` remains unspecified. Therefore, the effect of a statement such as `WriteLn(strval)` is unspecified.

You do not have to worry about this if you are dealing with the entire string. The statement

```
strval := 'F'
```

always works. The dynamic length of the string acquires a value of one, and the statement `WriteLn(strval)` prints an `F`. The values of character positions beyond position `1` are still unspecified, and the subsequent assignment

```
strval[2] := 'F'
```

leaves `strval` unchanged, because you cannot assign values in this way to character positions beyond the current length of the string.

The predefined procedures for string manipulation described in the "String Manipulation" chapter of the *Pascal Library Reference* always properly maintain the lengths of the string values they modify, and so are easier to use than this kind of indexed string manipulation.

A constant string may be accessed using an index expression. For example,

```
VAR
  x : CHAR;
BEGIN
  x := 'abcdef'[4];
  WriteLn(x);
END.
```

produces "d" as output.

The zeroth position of a string is special: it contains the dynamic length of the string.

For information on strings, see "String Types" on page 51.

Records and Field Designators

A specific field of a record variable is denoted by a variable access that refers to the record variable, followed by a field designator that specifies the field. The syntax for the field designator part looks like this:

<field designator> ::=

- <identifier>

Here are some examples of field designators:

```
employee.salary
coord.theta
```

The period (`.`), as well as the record variable identifier or function call, can be omitted inside a `WITH` statement that lists the record variable identifier or function call.

See also "The WITH Statement" on page 113.

File Window Variables

File-type variables have a special attribute which signifies the current position within the file, called the *current file position*. For some kinds of file variables, data objects located at the current file position can be accessed by using a special variable, the *file window variable*, which is obtained by applying a special qualifier to the file variable. This qualifier, which immediately follows the variable, is called the *file buffer symbol*:

<file buffer symbol> ::=

```
^
```

Thus the file window variable of file `F` is referred to by `F^`.

Use of the file window variable is discussed in the section "The File Window Variable" on page 185 and in the "Input and Output" chapter of the *Pascal Library Reference*.

Pointers and Their Identified Variables

The value of a pointer variable is either `NIL` or a value that identifies some other variable, called the *identified variable* of the pointer.

The identified variable of a pointer variable is accessed by using the pointer variable followed by a pointer symbol qualifier:

<pointer symbol> ::=

```
^
```

The constant `NIL` does not point to a variable. If you access memory by means of a `NIL` pointer reference, the results are unspecified. However, there may not be any error indication.

Here are some examples of references using pointers:

```
p1^
p1^.sibling^
```

Pointer types are discussed in "Pointer Types" on page 53.

6

Expressions

This chapter introduces the concept of an *expression*, and describes how expressions are used to designate values within CodeWarrior Pascal.

Expressions Overview

Every kind of *value* that is expressed in a Pascal program, whether determined at compile-time or at run-time, is designated by way of an *expression*.

Expressions can take one of two forms: An expression can be *atomic*, meaning that it consists of one single value component. A single variable access, for example, is an atomic expression. Alternatively, an expression can be *compound*, meaning that its value is a product of operations on and compositions of other expressions.

The first part of this chapter goes into a deeper explanation of atomic and compound expressions and how to use them. It covers the following topics:

- Atomic Expressions
- Compound Expressions
- Expression Syntax

The final part of the chapter is a comprehensive detailing of the various operators available for forming compound expressions:

- Operators

Atomic Expressions

All expressions designate values. The value of an atomic expression comes from a single value component, which can take one of three forms:

- A variable access
- An unsigned constant
- A function call

For details on variable accesses, see "Variables" on page 76. For details on constants, see "Constants" on page 73. For details on function calls, see "Invoking Functions" on page 145.

NOTE: Any number with a negation sign to the left of it (as in "-1") is not classified as an atomic expression. It is classified as a compound expression because it makes use of the unary negation operator. See "Compound Expressions" on page 89 for details about compound expressions and "Arithmetic Operators" on page 94 for details about the unary negation operator.

An atomic expression thus has the following abstract syntax:

<atomic expression> ::=

<variable access> | <unsigned constant> | <function call>

Constant Atomic Expressions

In a variable or constant initialization, values must be expressed in the form of a *constant expression*. Like all expressions, a constant expression can be either atomic or compound. A constant atomic expression is very similar to any other atomic expression. As you would expect, however, a variable access is not valid as a constant atomic expression. Here's the actual syntax:

<constant atomic expression> ::=

<unsigned constant> | <function call>

The only other constraint on a constant atomic expression is that only the following (constant) functions can be used within constant expressions:

```
Abs Chr Odd Ord Ord4 Round SizeOf Sqr Trunc
Upcase
```

For more information about these functions, see the *Pascal Library Reference*.

NOTE: When using SizeOf in constant expressions, only a single type or variable identifier is allowed. (The SizeOf function ordinarily allows field specifications.)

Compound Expressions

Compound expressions, like atomic expressions, designate some kind of value. The difference is that while an atomic expression designates its value as a single, atomic value component, a compound expression derives its value by performing operations which combine and transform any number of component expressions, or *subexpressions*, into a single value.

This section covers the following topics explaining in greater detail compound expressions:

- Operators and Operands
- Operator Precedence
- Constant Compound Expressions

Operators and Operands

In a compound expression, the one or more subexpressions are combined or transformed by way of an *operator*. Each operator is defined in terms how many subexpressions it can operate on and what operation it performs.

The expression or expressions that an operator performs upon are referred to as its *operands*. Each operator has its own restrictions as to which kinds of operands it can accept. Only operands with certain data types are compatible with each specific operator.

Each operator produces as a result a single value whose type depends on the operator itself as well as, often times, the operands involved in the expression. Since the result of combining one or more expressions this way is itself an expression, it is possible to build arbitrarily complex expressions by combining compound expressions with other compound expressions to get even more complex expressions.

The final section of this chapter is a listing of each kind of operator available in CodeWarrior Pascal. Along with each operator, it details which types of operands it can accept, as well as the type of the resulting value. See "Operators" on page 93.

Operator Precedence

When an expression contains multiple operators, operations are performed in order of their *precedence*. Table 6.1 shows the CodeWarrior Pascal operators and their precedences:

Table 6.1 Precedence of operators

Operators	Precedence	Category
`@`, `NOT`, `**`	highest	exponent and unary operators
`*`, `/`, `DIV`, `MOD`, `AND`, `&`	second	"multiplying" operators
`+`, —, `OR`, |	third	"adding" operators and signs
`=`, `<>`, `<`, `>`, `<=`, `>=`, `IN`	lowest	relational operators

Operations with equal precedence are performed from left to right, in general. The single exception is for the exponentiation operator

`(**)`. Multiple exponentiation operations are performed right to left.

Subexpressions which are not related by precedence may be evaluated in any order.

For detailed information about each operator, see "Operators" on page 93.

Constant Compound Expressions

Like constant *atomic* expressions, constant compound expressions designate constant values. Constant expressions (either atomic or compound) can be used in places where non-constant expressions can't… in variable and constant initializations, for instance.

The only difference between constant compound expressions and other compound expressions is that a *constant* complex expression is made up of subexpressions all of which are themselves constant expressions (either atomic or compound.) When an operator performs an operation on one or more constant expressions, the result is always itself a constant expression.

For information on constant atomic expressions, see "Constant Atomic Expressions" on page 88.

Expression Syntax

Below we specify the exact syntax for an expression in CodeWarrior Pascal. In order to capture the precedence of different operators, we introduce several auxiliary syntax objects, the *simple expression*, the *term*, and the *factor*.

<expression> ::=

<simple expression> [(**=** | **<** | **>** | **<=** | **>=** | **<>** | **IN**) <simple expression>]

<simple expression> ::=

[<sign>] <term> { (**+** | **-** | **OR** | **|**) <term> }

<term> ::=

<factor> { (***** | **/** | **DIV** | **MOD** | **AND** | **&** | ******) <factor> }

<factor> ::=

([**@**] <variable access>) | <atomic expression> | <set constructor> | **(** <expression> **)** | (**NOT** <factor>)

Constant Expression Syntax

The syntax for a constant expression is almost identical to that of other expressions, except that all subexpressions must also be constant expressions. The specification described in the following few diagrams uses the auxiliary syntax objects, the *simple expression*, the *term*, and the *factor*, to capture the precedence of the various operators:

<constant expression> ::=

<simple constant expression> [(**=** | **<** | **>** | **<=** | **>=** | **<>** | **IN**) <simple constant expression>]

<simple constant expression> ::=

[<sign>] <constant term> { (**+** | **-** | **OR** | **|**) <constant term> }

<constant term> ::=

<constant factor> { (***** | **/** | **DIV** | **MOD** | **AND** | **&** | ******) <constant factor>}

<constant factor> ::=

<constant atomic expression> | <set constructor> | **(** <constant expression> **)** | **NOT** <constant factor>

When using a constant expression in a type declaration that is declaring a subrange type, you cannot use a parenthesis as the first character after the equal sign. CodeWarrior Pascal distinguishes subrange types from enumerated types by the first symbol after the equal sign: a left parenthesis in that position signifies an enu-

merated type. If a subrange specification needs parentheses, precede it with “0+”. Here are two examples:

```
TYPE
  range =    0+(const1—const2) DIV 2..const2;
  color =    (brown, red, orange, yellow, green);
```

This rule only applies within the type declaration part of a program.

Operators

Operators allow complex expressions to be built up by performing operations on one or more subexpressions. For a more complete description of operators and complex expressions, see “Compound Expressions” on page 89.

This section contains a description each operator supported by CodeWarrior Pascal. Along with a description, a specification is given for each operator outlining what type of operands are compatible as well as the type of the resulting expression. For a definition of *operands*, see “Operators and Operands” on page 89.

Operators discussed in this section are organized by group. The groups discussed are as follows:

- Arithmetic Operators
- Boolean Operators
- String Operators
- Set Operators
- Relational Operators

In addition, there are two operators which defy categorization. They are:

- The @ Operator
- The Set Constructor

Arithmetic Operators

The types of operands and results for arithmetic binary and unary operations are shown in Table 6.2 and Table 6.3.

Table 6.2 Binary arithmetic operators

Operator	Operation	Operand type	Type of result
+	addition	INTEGER, LONGINT, or REAL type	INTEGER, LONGINT, or LONGREAL
–	subtraction	INTEGER, LONGINT, or REAL type	INTEGER, LONGINT, or LONGREAL
*	multiplication	INTEGER, LONGINT, or REAL type	INTEGER, LONGINT, or LONGREAL
**	exponentiation	INTEGER, LONGINT, or REAL type	INTEGER, LONGINT, or LONGREAL
/	division	INTEGER, LONGINT, or REAL type	REAL
DIV	division with integer result	INTEGER or LONGINT	INTEGER or LONGINT
MOD	remainder (to enable ANS Pascal-compatible behavior instead, see the *Pascal Compilers Reference*.)	INTEGER or LONGINT	INTEGER

The symbols +, –, and * are also used as set operators (described later in this chapter).

NOTE: Except for | and &, CodeWarrior Pascal may evaluate the operands of a binary operator in either order. With | and &, the left operand is evaluated first, then the right operand only if required to determine a value.

The real types are REAL or LONGREAL.

Table 6.3 Unary arithmetic operators (signs)

Operator	Operation	Operand type	Type of result
+	identity	INTEGER, LONGINT, or REAL type	INTEGER, LONGINT, or LONGREAL
–	sign negation	INTEGER, LONGINT, or REAL type	INTEGER, LONGINT, or LONGREAL

Any operand whose type is a subrange of a scalar type is treated as if it were of the scalar type.

If both the operands of an addition, subtraction, or multiplication operator are of type INTEGER or LONGINT, the result is of type INTEGER or LONGINT; otherwise, the result is of type LONGREAL.

The result of the identity or sign negation operator is of the same type as the operand, except that real types are converted to LONGREAL.

The value of *i* DIV *j* is the mathematical quotient of *i*/*j*, rounded toward zero to an INTEGER or LONGINT value. An error occurs if *j*=0.

The MOD operator returns the remainder of the division of its two operands. That is,

```
(–i )MOD j = –(i MOD j )
```

The value of *i* MOD *j* is equal to the value of

```
i –(i DIV j ) * j
```

The sign of the result of MOD is always the same as the sign of *i*. An error occurs if the value of *j* is zero.

NOTE: The name MOD is actually a misnomer; MOD returns the remainder after division of i by j. To obtain the modulus, a number in the closed interval from 0 to j – 1, use the expression

```
((i MOD j ) + j ) MOD j
```

In ANS Pascal, however, the actual modulus is returned. Since the CodeWarrior compiler can be configured to be ANS compliant, the `MOD` operator will have this modulus behavior if the compiler is in ANS-compliant mode. For information on how to set compiler options, see the "Pascal Settings" chapter of the Pascal Compilers Reference.

Boolean Operators

There are three types of boolean operators implemented in CodeWarrior Pascal.

- Logical Operators
- Bitwise Infix Operators

Logical Operators

The types of operands and results for logical operations are shown in Table 6.4.

Table 6.4 Logical operators

Operators	Left	Right	Result	Notes
`AND`	`BOOLEAN`	`BOOLEAN`	`BOOLEAN`	full evaluation
`&`	`BOOLEAN`	`BOOLEAN`	`BOOLEAN`	short circuit evaluation
`OR`	`BOOLEAN`	`BOOLEAN`	`BOOLEAN`	full evaluation
|	`BOOLEAN`	`BOOLEAN`	`BOOLEAN`	short circuit evaluation
`XOR`	`BOOLEAN`	`BOOLEAN`	`BOOLEAN`	always full evaluation
`NOT`		`BOOLEAN`	`BOOLEAN`	

Notice that there are two operators each for disjunction and conjunction. When you use the `AND` and `OR` operators, the compiler evaluates the entire expression, even if that is not necessary. For example, consider the expression

```
TRUE OR boolTst(x)
```

where `boolTst` is a function that returns a `boolean` value. This expression will always have the value `true`, regardless of the result of `boolTst(`*x*`)`. However, `boolTst(`*x*`)` will always be called. This could be important if `boolTst` has side effects.

With the expression

```
TRUE | boolTst(x)
```

evaluation stops as soon as the value `true` is reached, because evaluation proceeds from left to right for operators of the same precedence. The ampersand (`&`) and vertical bar (`|`) are referred to as the *short-circuit operators*.

NOTE: CodeWarrior Pascal does not support mixing short-circuit operations and normal operators in the same expression.

Bitwise Infix Operators

The types of operands and results for bitwise infix operations are shown in Table 6.5.

Table 6.5 Bitwise Infix Operators

Operators	Left	Right	Type of Result
AND	INTEGER, LONGINT	INTEGER, LONGINT	INTEGER, LONGINT
OR	INTEGER, LONGINT	INTEGER, LONGINT	INTEGER, LONGINT
XOR	INTEGER, LONGINT	INTEGER, LONGINT	INTEGER, LONGINT
SHL	INTEGER, LONGINT	INTEGER, LONGINT	INTEGER, LONGINT
SHR	INTEGER, LONGINT	INTEGER, LONGINT	INTEGER, LONGINT
NOT		INTEGER, LONGINT	INTEGER, LONGINT

For bitwise infix operators, the operands are integer based types. The result type is the common type of both operands using integer promotion.

String Operators

The plus sign (+) can be used for string concatenation, like the `Concat` routine in the Pascal runtime library.

Listing 6.1 Example of string concatenation using +

```
VAR
  a, b, c, d : STRING[50];

BEGIN
  a := 'The ';
  b := 'green ';
  c := 'cat.'
  d := a + b + c; { d gets 'The green cat.' }
END;
```

Set Operators

The types of operands for set operations are shown in Table 6.6.

Table 6.6 Set operators

Operator	Operation	Operand type
+	union	compatible set types
–	difference	compatible set types
*	intersection	compatible set types

Result types in set operations

The following rules govern the type of the result of a set operation where one or both of the operands are a `SET OF subr`. In the statement of these rules, `ordtyp` represents any scalar type and `subr` represents a subrange of `ordtyp`:

- If `ordtyp` is not type `INTEGER`, then the type of the result is type `SET OF ordtyp`.

- If `ordtyp` is type integer, then the type of the result is type `SET OF 0..255`. This rule results from the limitations described under "Set Types" on page 61.

Relational Operators

The types of operands and results for relational operations are shown in Table 6.7.

Table 6.7 Relational operators

Operator	Operation	Operand type	Type of result
=	equal to	compatible set, simple, or pointer types	BOOLEAN
<>	not equal to	compatible set, simple, or pointer types	BOOLEAN
<	less than	compatible simple types	BOOLEAN
>	greater than	compatible simple types	BOOLEAN
<=	less than or equal to	compatible simple types	BOOLEAN
>=	greater than or equal to	compatible simple types	BOOLEAN
<=	subset of	compatible set types	BOOLEAN
>=	superset of	compatible set types	BOOLEAN
IN	member of	left operand: any scalar type `T`; right operand: type `SET OF T`	BOOLEAN

Comparing Numbers

When the operands of =, <>, <, >, >=, or <= are numeric, they need not be of compatible type if one operand is of real type and the other is INTEGER or LONGINT.

NOTE: On a Macintosh, because of extensions provided for use with the Standard Apple Numeric Environment (SANE), the result of a comparison can be unordered. An unordered result occurs from a comparison involving a NaN (Not a Number). One important effect is that `NOT(a<b)` is true if either `a` is greater than `b` or `a` and `b` are unordered. You can use the `Relation` function, which is available through the SANE.p Universal interface, to test for an unordered comparison.

Comparing Booleans

If *p* and *q* are `BOOLEAN` operands, then *p* = *q* denotes their equivalence, *p* <> *q* denotes the logical exclusive-or operation, and *p* <= *q* denotes the logical expression "*p* implies *q*" (because `FALSE<TRUE`). You can also write `NOT` *p* `OR` *q* for logical implication.

Comparing Strings

When the relational operators =, <>, <, >, <=, and > are used to compare strings, they order them according to the ordering of the ASCII character set. Note that any two string values can be compared because all string values are compatible. String comparisons follow these steps:

1. **The two strings are compared a character at a time, starting with the first character.**
2. **Two corresponding characters are compared.**

   If the ASCII value of one character is greater than the other, then the corresponding string is greater than the other.
3. **If the two corresponding characters are equal, the point of comparison advances to the next character in each string, and the process returns to step 2.**
4. **If the end of one string has been reached, its value is less than the other string.**
5. **If the ends of both strings have been reached, the two strings are equal.**

NOTE: If developing for the Mac OS platform, a set of utilities that apply ordinary language rules for comparing strings is available as an operating system service. If you use those utilities to compare strings rather than using the operators described here, alphabetization will follow the local language's rules rather than the ASCII table. These routines are documented in the primary Mac OS Toolbox documentation - Inside Macintosh - in the "Text" volume.

Comparing Sets

If *u* and *v* are set operands, their comparisons have these results:

- *u* <= *v* is true if *u* is included in *v*.
- *u* >= *v* is true if *v* is included in *u*.
- *u* = *v* is true if *u* and *v* contain exactly the same members.
- *u* <> *v* is true if either *u* or *v* contains a member not contained in the other.

Testing Set Membership

The `IN` operator yields the value `true` if the value of the scalar type operand is a member of the set type operand; otherwise, it yields the value `FALSE`.

Comparing PACKED ARRAY OF CHAR

In addition to the operand types shown in the table, the = and <> operators can also be used to compare a `PACKED ARRAY`[*m*..*m* + *n* – 1] `OF CHAR` with a string constant containing exactly *n* characters, or to compare two one-dimensional `PACKED ARRAYs OF CHAR` of identical type. The comparison follows the steps given above under "Comparing Strings" on page 100.

The @ Operator

A pointer to a variable (an address) can be computed with the `@` operator. The operand and result types are shown in Table 6.8.

Table 6.8 The pointer operator

Operator	Operation	Operand	Type of result
`@`	pointer formation	variable, parameter, procedure, or function	pointer

The `@` operator is a unary operator taking a single variable, parameter, procedure, or function as its operand and computing the value of its pointer. The type of the value is equivalent to the type of `NIL` and consequently can be assigned to any pointer variable.

The section that originally followed was really Mac-specific and so I removed it in favor of the following warning:

WARNING! The actual storage implementations of variables, parameters, procedures and functions impose certain limitations on the usage of the `@` operator as well the resulting pointer value. Details can be found in the appropriate Targeting manual and the Pascal Compilers Reference.

The Set Constructor

The set *constructor* is a kind of operator with syntax different from all other operators. It is used to construct set type values out of operands from the set's base type. Set constructors are formed by writing expressions within square brackets, as shown in the syntax diagrams below:

<set constructor> ::=

[[<member group> { **,** <member group> }] **]**

<member group> ::=

<expression> [**..** <expression>]

Table 6.9 The set constructor

Operator	Operation	Operand	Type of result
`[]`	set construction	base type of given set type	set type

All values designated in member groups in a particular set constructor must be of the same scalar type. This scalar type is the base type of the resulting set.

The notation `[]` denotes the empty set, which belongs to every set type. Any member group *x*..*y* denotes as set members the range of all values of the base type in the closed interval *x* to *y*. If *x* is greater than *y*, then *x*..*y* denotes no members and `[`*x*..*y*`]` denotes the empty set.

Here are some examples of set constructors (Listing 6.2):

Listing 6.2 Example Set Constructors

```
[red, c, green]
[1, 5, 10..k MOD 12, 23]
['A'..'Z', 'a'..'z', chr(xcode)]
```

7

Statements

This chapter describes the notion of a statement, and then goes on to outline the various kinds of statements provided by CodeWarrior Pascal.

Statements Overview

Statements are the *actions* of the Pascal language. By combining special sequences of reserved words, variables, and expressions, statements perform the actual execution-time tasks of a Pascal program, controlling the state of the program data and as well as its path of execution.

Statements are always found within Pascal *blocks*, which are the fundamental executable chunks of a Pascal program. To find out more about what blocks are and how they work, see "Blocks and Scope Overview" on page 161.

The first part of this chapter discusses three types of statements:

- Repetition Statements
- Conditional Statements
- Control Statements

The chapter then moves on to describe, in detail, each individual statement supported by CodeWarrior Pascal:

- The Assignment Statement
- The Compound Statement
- The Procedure Statement
- The WITH Statement
- The FOR Statement
- The WHILE Statement

- The REPEAT Statement
- The IF Statement
- The CASE Statement
- The GOTO Statement
- The RETURN Statement
- The CYCLE Statement
- The LEAVE Statement
- The Null Statement

Repetition Statements

Pascal provides three ways to execute the same program section repeatedly—the process called *looping*. Pascal sets up the loop and exit routines for you; all you need to do is specify the conditions for repetition. The repetition statements are the following:

- The `FOR` statement `FOR...DO` executes the same program section a given number of times. The number of executions may be constant or may be determined by the result of any scalar calculation. See "The FOR Statement" on page 114 for more information.
- The `WHILE` statement `WHILE...DO` executes the same program section repeatedly as long as a given boolean expression is true. It evaluates the boolean control before each pass, including the first time; hence it can bypass the program section altogether. See "The WHILE Statement" on page 116 for more information.
- The `REPEAT` statement `REPEAT...UNTIL` also executes the same program section repeatedly as long as a given boolean expression is true. But it evaluates the boolean control after each pass; hence it executes the program section at least once. See "The REPEAT Statement" on page 117 for more information.

Loop Control: A Comparison

The three repetition statements each have specific advantages and disadvantages in any given programming situation. Here are some of them.

The `FOR` statement automatically keeps track of which repetition it is executing, by changing the value of its control variable at the end of each pass. Thus you can use the control value to modify what your program does each time. For example, the control value can cause the repeated section to:

- Select a different element in an array each time by changing the index number.
- Call a different procedure each time by serving as the selector value for the `CASE` statement (described below.)
- Perform a different calculation each time by becoming a factor in an expression.

The `FOR` statement is somewhat inflexible, however. You can change its number of repetitions only by terminating it with a `GOTO` or `LEAVE` statement.

The `WHILE` statement and `REPEAT` statement allow better control of the conditions under which they stop executing. The main difference between them is that the `WHILE` statement need not be executed at all, whereas the `REPEAT` statement executes at least once. Thus the `WHILE` statement is most useful when the condition controlling its execution may have already been satisfied; the `REPEAT` statement is most useful when the condition can be satisfied only by executing the statement.

The `WHILE` statement should also be used in cases where executing it under the wrong conditions could be detrimental, because it evaluates its control before each pass.

Conditional Statements

Pascal provides two ways for your program to choose what to do next—the process sometimes called *branching*:

- The `IF` statement `IF...THEN...ELSE` evaluates a boolean expression and executes a controlled statement only if it is true. It can also be written to execute a second statement if the expression is false. See "The IF Statement" on page 118 for more information.
- The `CASE` statement `CASE...OF...OTHERWISE` executes one statement from a list, depending on the value of a scalar control expression. See "The CASE Statement" on page 120 for more information.

Control Statements

The three repetition statements and two conditional statements described in this chapter, along with assignments and procedure calls, are flexible enough to handle almost all programming jobs. Occasionally, however, you may encounter a situation that demands immediate transfer or suspension of program execution. For these cases, Pascal provides five additional tools:

- The `GOTO` statement, which transfers control directly from one program statement to another. See "The GOTO Statement" on page 121 for more information.
- The `CYCLE` directive, which forces an immediate reiteration of a repetition statement loop. See "The CYCLE Statement" on page 122 for more information.
- The `LEAVE` directive, which immediately cancels a repetition statement loop. See "The LEAVE Statement" on page 123 for more information.
- The null statement, which does nothing, is often used in loops. See "The Null Statement" on page 124 for more information.
- The `Exit` procedure, which terminates any procedure, function, or whole program. See the "Exit and Halt" chapter of the *Pascal Library Reference* for more information.
- The `Halt` procedure, which stops program execution then and there. See the "Exit and Halt" chapter of the *Pascal Library Reference* for more information.

Individual Statement Descriptions

This topic describes each statement supported by CodeWarrior Pascal. The following statements are discussed

- The Assignment Statement
- The Compound Statement
- The Procedure Statement
- The WITH Statement
- The FOR Statement
- The WHILE Statement
- The REPEAT Statement
- The IF Statement
- The CASE Statement
- The GOTO Statement
- The CYCLE Statement
- The LEAVE Statement
- The Null Statement

The Assignment Statement

The assignment statement sets the value of a variable. The symbol **:=** can be read as "set to." Here is what the syntax of an assignment statement looks like:

<assignment statement> ::=

(<variable access> | <function identifier>) **:=** <expression>

The variable reference of the left side identifies a variable of any of the types except a file type. With most variables it is simply an identifying name, but in four cases it consists of a name followed by a qualification:

- If the variable is a string element, it is identified by the string name followed by the element's index number in brackets.

- If the variable is an array element, it is identified by the array name followed by one or more index values (one for each dimension of the array) enclosed in square brackets and separated by commas.
- If the variable is a record field, its name must be preceded by the name of its containing record and a period (unless the assignment statement is enclosed in a `WITH` statement). For details on using the `WITH` statement, see "The WITH Statement" on page 113.
- If the variable is a dynamic variable, it is identified by the name of its pointer followed by a caret.

In writing assignments, keep these rules in mind:

- A real variable may be set to the value of another `REAL`, `SINGLE`, `DOUBLE`, an `INTEGER` or `LONGINT`, `UNSIGNEDWORD`, `UNSIGNEDLONG`, an integer subrange type, or an expression yielding integer or real results.
- A `LONGINT` variable may be set to the value of an `INTEGER`, `UNSIGNEDWORD`, `UNSIGNEDLONG`, an `INTEGER` subrange type, or an expression yielding integer results. It may be set to the value of another `LONGINT` or to an expression yielding `LONGINT` results, provided the actual value does not exceed its declared size.
- A `BOOLEAN` variable may be set to the value of another `BOOLEAN` or to an expression yielding a `BOOLEAN` result.
- A `CHAR` variable may be set to the value of another `CHAR`, a `CHAR` subrange type, or a `STRING` element.
- A scalar variable of subrange type may be set to the value of another scalar (or an expression yielding scalar results) of the host type, provided the actual value lies within its declared range.
- A user-defined scalar variable may be set to any of the values named in its declaration.
- A string variable may be set to the value of another string, provided its actual length does not exceed the variable's declared size.
- A set variable may take the value of another set variable or set constructor, provided they have the same base type.

- A whole array variable or record variable may be set to the value of another whole array or record variable of assignment-compatible type.
- A one-dimensional packed character array variable may be set to the value of a string constant (but not a string variable), provided its index range is the same as the string's length.
- Array elements (including elements that are arrays) and record fields (including fields that are records) act in assignments like ordinary variables of their declared types.
- The right-hand side of an assignment statement is always evaluated before the left-hand side.

Here are some examples of assignment statements (Listing 7.1):

Listing 7.1 Example Assignment Statements

```
x := y+z
p := (1<=i) AND (i<100)
i := Sqr(k)–(i*j)
hue1 := [blue, succ(c)]
```

The Compound Statement

When writing Pascal programs, you often need to treat several statements as if they were one—for example, when they are all executed by a single control statement. To do this, you use the compound statement.

The body of every Pascal procedure, function, and main program consists of a single compound statement. To create a single compound statement out of a sequence of statements, preface the sequence with `BEGIN` and terminate it with `END`, separating the internal statements with semicolons:

<compound statement> ::=

`BEGIN` <statement> {**`;`** <statement>} **`END`**

You can nest any number of `BEGIN...END` statements. Within any block, the compiler will associate the last `BEGIN` with the first `END`,

the next-to-last `BEGIN` with the second `END`, and so on. If you have written more `BEGIN`s than `END`s, the compiler will stop and display an error message.

Here is an example of a compound statement (Listing 7.2):

Listing 7.2 Example Compound Statement

```
BEGIN
  z := x;
  x := y;
  y := z
END
```

The Procedure Statement

A procedure is called by writing its identifier in the source text, followed by its actual parameter list (if it has one), as shown in the following syntax diagrams:

<procedure statement> ::=

<procedure identifier> [<actual parameter list>]

<actual parameter list> ::=

([<actual parameter> { **,** <actual parameter>}] **)**

<actual parameter> ::=

<expression> | <variable access> | <procedure identifier> | <function identifier>

The identifier must be the same as the identifier used in the procedure or function declaration. For information on declaring procedures, see "Procedure Declarations" on page 126.

For a detailed discussion of actual parameters and their relationship to the *formal parameters* as declared in the procedure definition, see "Parameters" on page 133.

Here are some examples of procedure statements (Listing 7.3):

Listing 7.3 Example Procedure Statements

```
printheading;
transpose(a, n, m);
bisect(fct, –1.0, +1.0, x);
```

The WITH Statement

The WITH...DO statement provides a means by which the fields of specified records can be referenced using only their field identifiers:

<WITH statement> ::=

WITH (<record variable access> | <object reference variable access>) { **,** (<record variable access> | <object reference variable access>) } **DO** <statement>

Any number of record variable identifiers, including those of records that are fields of other records, may be listed between WITH and DO. The statement

```
WITH v1, v2, v3 DO s;
```

is equivalent to the group of WITH statements

```
WITH v1 DO
  WITH v2 DO
    WITH v3 DO s
```

The following rules govern the use of WITH...DO:

- When listing a record that is a field of another record, you must either list the containing record earlier or list that field in explicit form.
- WITH statements may be nested. The record variables "opened" by any WITH statement remain open in the nested statements.
- Where fields of different record variables have the same name, WITH accesses the field of that name in the record last listed, including redundant listings in nested statements. The identity of field names does not cause a compiler error.

- Where a record field identifier is the same as a variable or other identifier declared outside the record, `WITH` accesses the field.
- Within a `WITH` statement, fields may still be identified explicitly, even though their record variables are listed. This feature can be used to resolve the ambiguity of identical field names.
- When used with variant record variables, `WITH` accesses the identifiers for their tag fields and all variant fields.

Here is an example of a `WITH` statement (Listing 7.4):

Listing 7.4 Example WITH Statement

```
WITH date DO IF month=12 THEN
 BEGIN
  month := 1;
  year := year+1
 END
 ELSE month := month+1;
```

The FOR Statement

The `FOR` statement requires a previously declared local variable of scalar type. It repeatedly increments or decrements the value of the variable, executing a section of your program each time. You define the starting and ending scalar values (which may be constant or calculated) and whether the `FOR` statement is to count upward or downward. The following syntax diagrams describe the actual syntax of the `FOR` statement:

<FOR statement> ::=

FOR <control variable> **:=** <initial value> (**TO** | **DOWNTO**) <final value> [**BY** step value] **DO** <statement>

<control variable> ::=

<variable identifier>

<initial value> ::=

<expression>

<final value> ::=

<expression>

<step value> ::=

<constant>

The control variable is the name of a scalar variable—INTEGER, CHAR, BOOLEAN, subrange, or user-defined. It cannot be an array or string element, a record field, or a dynamic variable. It must be declared in the block that contains the FOR statement. The FOR statement gives it a value before each pass through the program section it controls. Note that the value of this variable is accessible in the controlled section.

The initial and final value expressions must have the same scalar type as the variable. They may be simple constants or variables, or complex expressions containing operators and functions.

You write TO or DOWNTO, depending on whether the ordinality of the value of the second expression is higher or lower than the ordinality of the value of the first expression.

You write BY to increment or decrement the loop by a constant other than 1. The step expression must be of the type INTEGER. BY is considered a reserved word in this case only.

The statement controlled by the FOR statement can be a single other statement (such as an assignment or a procedure call) or a compound statement containing many other statements.

Observe these rules and cautions when writing any FOR statement:

- The control variable must be a simple variable with local scope.
- If the control variable is a subrange type or user-defined scalar, it must be capable of accepting the initial and limit values as well as all values with an ordinality in between.

- Do not try to change the value of the control variable from within the FOR statement; doing so can have unpredictable results.
- Do not include the control variable in either of the limit expressions.
- After the FOR statement is finished, the value of the control variable may be unspecified.
- The limit expressions are evaluated just once, before the first pass. Changing them from within the FOR statement will not alter its behavior.
- If the limit expressions have equal value, the FOR statement will execute its controlled statement once.
- If the limit values are reversed—large limit less than small limit—the FOR statement will be skipped.

Here are some examples of FOR statements (Listing 7.5):

Listing 7.5 Example FOR Statements

```
FOR i := 5 TO 115 STEP 5 DO
  IF a[i]>max THEN max := a[i];

FOR i := 1 TO n DO FOR j := 1 TO n DO
  BEGIN
    x := 0;
    FOR k := 1 TO n DO x := x+m1[i, k]*m2[k, j];
    m[i, j] := x
  END;

FOR c := red TO blue DO q(c);
```

The WHILE Statement

The WHILE statement evaluates a boolean expression and then executes a statement if the expression is true. It repeats the execution, evaluating the expression before each pass, until the expression becomes false. Here is what the syntax for the WHILE statement looks like:

<WHILE statement> ::=

WHILE <expression> **DO** <statement>

The controlling expression must have `boolean` type; usually it is formed out of relational and logical operators.

The statement controlled by `WHILE...DO` may be either a single statement or a compound `BEGIN...END` construction containing other statements.

Here are some examples of `WHILE` statements (Listing 7.6):

Listing 7.6 Example WHILE Statements

```
WHILE a[i]<>x DO i := i+1;

WHILE i>0 DO
  BEGIN
    IF Odd(i) THEN z := z*x;
    i := i DIV 2;
    x := sqr(x)
  END;

WHILE NOT EOF(f) DO
  BEGIN
    process(f^);
    get(f)
  END;
```

The REPEAT Statement

The `REPEAT` statement behaves much like the `WHILE` statement, but it evaluates its boolean expression after executing the statements it controls. Its syntax is as follows:

<REPEAT statement> ::=

REPEAT <statement> { **;** <statement> } **UNTIL** <expression>

REPEAT and UNTIL create their own compound out of the statements they control; you do not need to use BEGIN and END.

The controlling expression must have BOOLEAN type; usually it is formed out of relational and logical operators.

NOTE: With both WHILE and REPEAT, take care that the program statements they control either change the value of the control expression or escape by means of a GOTO or LEAVE statement or Exit call. Otherwise your program will never be able to terminate.

Here are some examples of REPEAT statements (Listing 7.7):

Listing 7.7 Example REPEAT Statements

```
REPEAT
  k := i MOD j;
  i := j;
  j := k
UNTIL j=0;

REPEAT
  Process(f^);
  Get(f)
UNTIL EOF(f);
```

The IF Statement

The IF statement executes a single controlled statement (which may be a compound BEGIN...END construction) if a boolean expression is true. You can add an optional ELSE part on the end that executes another (possibly compound) statement if it is FALSE:

<IF statement> ::=

IF <expression> **THEN** <statement> [**ELSE** <statement>]

The controlling expression between IF and THEN must have boolean type; usually it is formed out of relational and logical operators.

Either or both controlled statements may be single statements or compound BEGIN...END constructions containing other statements. The only place you need to put a semicolon in an IF statement is within a compound BEGIN...END construction.

When executing an IF statement, Pascal performs these steps:

1. **It evaluates the boolean expression.**
2. **If its value is true, Pascal executes the statement following THEN and exits the IF statement.**
3. **If its value is false and there is a statement after ELSE, Pascal executes it; otherwise it exits the IF statement.**

Here are some examples of IF statements (Listing 7.8):

Listing 7.8 Example IF Statements

```
IF x<1.5 THEN z := x+y ELSE z := 1.5;
IF p1<>nil THEN p1 := p1^.father;
```

Nested IF Statements

In any IF statement, the statement following the word ELSE can also be an IF statement and can contain its own ELSE clause. Thus an IF statement can be written to take different actions for each of several mutually exclusive conditions.

Pascal will evaluate boolean expressions only until a true one is found. You get maximum execution speed if you put the most probable conditions first.

The statement following the word THEN can also be a nested IF statement, but this can create confusing source text. If it becomes unclear which ELSE matches which THEN, clarify the situation by using a compound BEGIN...END construction or appropriate indentation.

The CASE Statement

The CASE statement lets you write a list of alternative statements to be executed, associating a scalar constant with each one. When executing the CASE statement, Pascal evaluates a controlling scalar expression; if its value matches one of the constants, Pascal executes the corresponding statement. You can add an optional OTHERWISE part on the end that executes a different statement in the case that nothing was selected from the list. The CASE statement follows this syntax described in the following diagrams:

<CASE statement> ::=

CASE <expression> **OF** <case> { **;** <case>} [<otherwise clause>] [**;**] **END**

<case> ::=

<constant range> { **,** <constant range> } **:** <statement>

<otherwise clause> ::=

; OTHERWISE <statement>

The controlling expression may have any scalar type—INTEGER, CHAR, BOOLEAN, subrange, or user-defined. It should be capable of returning the value of any of the constants in the CASE clause.

The constant expressions in the CASE clause must have the same scalar type as the controlling expression.

Any of the controlled statements in the CASE clause or the default statement following OTHERWISE may be single statements or compound BEGIN...END constructions containing other statements.

Here are two examples of CASE statements (Listing 7.9):

Listing 7.9 Example CASE Statements

```
CASE operator OF
  plus:  x := x+y;
  minus: x := x-y;
  times: x := x*y
```

```
END;

CASE i OF
  1:         x := sin(x);
  2:         x := cos(x);
  3, 4, 5:   x := exp(x);
  OTHERWISE x := ln(x)
END;
```

The GOTO Statement

The GOTO statement transfers program execution to the beginning of any statement that is within the same procedure, function, or main program. Before you can use a GOTO statement, you must do two things:

- Declare a label for every GOTO destination in your program. The label declaration consists of the reserved word LABEL, followed by one or more labels separated by commas.The destination label is an identifier. Label declarations are discussed under "Block Structure" on page 161.
- Write one of the declared destination labels, followed by a colon, in front of the statement that is the destination for each GOTO statement.

The syntax for the GOTO statement is described by the following syntax diagram:

<GOTO statement> ::=

GOTO <label>

Two more cautions apply to GOTO statements:

- The destination of any GOTO statement must be the beginning of a statement.
- Jumping to a statement that is within the structure of another statement (except within a compound statement that forms a program block) can have undefined effects, although the compiler will not indicate an error.

Thus every GOTO destination should be the beginning of a statement that is at the top level of nesting in a program block.

Here is an example illustrating the use of a GOTO statement (Listing 7.10):

Listing 7.10 Example GOTO Statement

```
BEGIN
  1234: Write ('Give me a number: ');
  Readln(n);
  IF n=0 THEN GOTO 1234
END;
```

The RETURN Statement

The RETURN statement jumps to the end of the current routine, skipping all subsequent statements in the routine and allowing execution to return to the calling routine. It uses the syntax:

<RETURN statement> ::=

RETURN { <expression>}

In a function, the expression in a RETURN statement specifies the function's result, making it unnecessary to assign the function's identifier with the return value.

NOTE: The type of a function's return value may not be a schema. See "Schemata" on page 64 for more information.

If a procedure, the expression in a RETURN statement is ignored, and the program returns execution at the procedure's calling routine.

The CYCLE Statement

The CYCLE directive passes program control to the end of the looping portion of the smallest WHILE, REPEAT, or FOR statement that

encloses it. It is similar to the `continue` statement in the C/C++ language. It uses the following syntax:

<CYCLE statement> ::=

CYCLE

Here is an example illustrating the use of a `CYCLE` directive. It calls the procedure `f` for all positive values of `a[i]` (Listing 7.11):

Listing 7.11 Example of CYCLE

```
FOR i := 1 TO n DO
  BEGIN
    IF a[i]<=0 THEN CYCLE;
    f(a[i])
  END;
```

The LEAVE Statement

The `LEAVE` directive terminates the smallest `WHILE`, `REPEAT`, or `FOR` statement that encloses it, passing control to the next statement. Its syntax looks as follows:

<LEAVE statement> ::=

LEAVE

Here is an example illustrating the use of a `LEAVE` directive; in it, the `WHILE` statement terminates when the first `x` value of `a[i]` is found (Listing 7.12):

Listing 7.12 Example of LEAVE

```
WHILE i<63 DO
  BEGIN
    IF a[i]=x THEN LEAVE;
    i := i+1
  END;
```

The Null Statement

Null statements are statements that don't contain anything. This simply means that whenever Pascal syntax calls for a statement, you can omit it. It also means that when a program contains an unnecessary semicolon, the Pascal compiler considers the semicolon to be separating a null statement from another statement. The result is two statements where you only intend one. Most of the time, this is harmless, but it occasionally causes an error when only one statement is allowed.

Listing 7.13 Example Null Statement

```
REPEAT
  ; { A null statement }
UNTIL ButtonPressed();
```

8

Procedures & Functions

This chapter describes how procedures and functions can be defined and used within Pascal programs.

Procedures and Functions Overview

By using procedures and functions, it is possible to make use of externally defined blocks of code from within other Pascal code. This capability is important in terms of factoring large Pascal programs into smaller, more manageable chunks, as well as the factoring of useful code into routines which can be re-used both within a single Pascal program as well as between multiple programs.

The rest of this chapter discusses the following topics pertaining to procedures and functions:

- Declaring Procedures and Functions
- Parameters
- Invoking Procedures and Functions
- Using C Routines Within Pascal

Declaring Procedures and Functions

Procedure and function identifiers must be declared within a block or unit in order to be invoked from within that block or unit. In addition to making this association between an identifier and a a procedure or function, the actual block of code to be executed upon invocation of the routine is also specified along with the declaration. Alternatively, certain *directives* can be applied to the procedure or

function declaration which tell the compiler where to find the actual executable code if it is not included in the declaration.

This section covers the following topics pertaining to procedure and function declarations:

- Procedure Declarations
- Function Declarations
- Procedure And Function Directives

For information on where procedure and function declaration fit into Pascal blocks, see "Blocks and Scope Overview" on page 161; to see how they are used in Pascal units, see "Programs and Units Overview" on page 169.

Procedure Declarations

A procedure declaration associates an identifier with a block of code, so the procedure block can be invoked from within another block of pascal code.

Here is the syntax for a procedure declaration:

<procedure declaration> ::=

<procedure heading> **;** <procedure body> **;**

The procedure heading specifies the identifier for the procedure and its formal parameters (if any), as shown in the following syntax diagram (the syntax for a formal parameter list is discussed later in this chapter in the section "Parameters" on page 133):

<procedure heading> ::=

PROCEDURE [<object type identifier> **.**] <identifier> [<formal parameter list>]

The procedure body specifies the actual code to be executed when the procedure is called (it can also specify a *procedure directive*:, this is discussed in the section "Procedure And Function Directives" on page 129):

<procedure body> ::=

([**ASM;**] <block>) | (**FORWARD**) | ([**C;**] **EXTERNAL**) | (**IN-LINE** <constant expression> { **,** <constant expression>})

An object type is only given if this is a method declaration. A method declaration is a special type of procedure declaration made as part of an object type declaration.

Here is an example of a procedure declaration (Listing 8.1):

Listing 8.1 Example Procedure Declaration

```
PROCEDURE Summation (n: INTEGER;
                     a: intarray;
                     VAR sum: LONGINT);
  VAR i: INTEGER;
  BEGIN {Summation}
    Sum := 0;
    FOR i := 1 TO n DO sum := Sum+a[i]
  END; {Summation}
```

For more information on method declarations, see "Object Pascal Overview" on page 147..

Function Declarations

A function declaration associates an identifier with a block of code that returns a value to the caller of the function. The return value can be of any type. Here is the syntax for a function declaration:

<function declaration> ::=

<function heading> **;** <function body> **;**

The function heading specifies the identifier for the function, the formal parameters (if any), and the type of the function result, as illustrated in the next syntax diagram (the syntax for a formal parameter list is discussed later in this chapter in the section "Parameters" on page 133):

<function heading> ::=

FUNCTION [<object type identifier> **.**] <identifier> [<formal parameter list>] **:** <type identifier>

The function body specifies the actual code to be executed when the procedure is called (it can also specify a *procedure directive*:, this is discussed in the section "Procedure And Function Directives" on page 129):

<function body> ::=

([**ASM;**] <block>) | (**FORWARD**) | ([**C;**] **EXTERNAL**) | (**INLINE** <constant expression> { **,** <constant expression> })

The statements to be executed upon activation of the function are specified by the statement part of the function's block.

So that the function may return a result, this block should either

- contain at least one assignment statement that assigns a value to the function identifier.

  The result of the function is the last value assigned.
- have a `RETURN` statement.

If no assignment statement for the function's identifier exists, if it exists but is not executed, or any `RETURN` statements in the function are not executed, the value returned by the function is unspecified. If the return value of the function is a structured type, you can assign values to components alone or to the entire structure.

NOTE: If the return value of a function is a record type, a pointer to a record type, you cannot use a `WITH` statement to assign values to the fields of the record. CodeWarrior Pascal will interpret use of the function's identifier in the `WITH` statement as a function call.

Here are some examples of function declarations (Listing 8.2):

Listing 8.2 Example Function Declarations

```
FUNCTION max(a: vector; n: INTEGER): LONGREAL;
  VAR x: LONGREAL; i: INTEGER;
  BEGIN
    x := a[1];
    FOR i := 2 TO n DO if x < a[i] THEN x := a[i];
    max := x
  END;

FUNCTION power(x: LONGREAL; y: INTEGER): LONGREAL; { y >= 0}
  VAR w, z: LONGREAL; i: INTEGER;
  BEGIN
    w := x; z := 1; i := y;
    WHILE i > 0 DO BEGIN
      {z*(w**i) = x**y}
      IF Odd(i) THEN z := z*w;
      i := i DIV 2;
      w := sqr(w)
    END;
    {z = x**y}
    power := z
  END;
```

For more information on the RETURN statement, see "The RETURN Statement" on page 122.

Procedure And Function Directives

In place of the block in a procedure or function declaration, you can write the following directives:

- FORWARD lets you use the procedure or function immediately but postpone defining the block to a later part of your program. See "FORWARD Directive" on page 130 for more information.
- EXTERNAL and C let you link a C/C++ or assembly-language routine to your program, which will be executed as the procedure or function's block. See "The EXTERNAL and C Directives" on page 132 for more information.

- When compiling coder to run on a 68k Macintosh, the `INLINE` directive lets you write actual assembly-language instructions to be executed in place of the block. It is not valid for other platforms. See the "68k Notes" chapter of the *Pascal Compilers Reference* for more information.
- When compiling code to run on a Macintosh, the `FAR` directive lets you access a routine or variable in far mode (for the 68k architecture only). It is not valid for other platforms. See the "68k Notes" chapter of the *Pascal Compilers Reference* for more information.
- `ASM` lets you write assembly language code within the definition of a procedure or function. See "The ASM directive" on page 133 for more information.

FORWARD Directive

A procedure or function declaration containing the directive `FORWARD` instead of a block is called a *forward declaration*. Somewhere after the forward declaration but in the same block, the procedure or function is defined by a *defining declaration*—a declaration that uses the same identifier and includes a block. The formal parameter list may be repeated in the defining declaration; but if you repeat the formal parameter list, it must be identical to the list in the forward declaration. The forward declaration and the defining declaration must be local to the same block but need not be contiguous; that is, other procedures or functions can be declared between them and can call the procedure that has been declared forward. This permits the forward declaration of classes and mutual recursion.

The forward declaration and the defining declaration constitute a complete declaration of the procedure or function. The procedure or function is considered to be declared at the place of the forward declaration.

Forward declaration of objects

When declaring an object that contains a variable of another object type that must be defined later in the same source code file, use the `FORWARD` directive. Objects declared with `FORWARD` cannot be inherited from.

Listing 8.3 Example of FORWARD Declaration for objects

```
objA = OBJECT; FORWARD;
objB = OBJECT
  ...
  o : objA; { objB uses objA }
  ...
END;

objA = OBJECT
  ...
  p : objB; { objB uses objA }
  ...
END;
```

For information on objects, see "Object Pascal Overview" on page 147.

Mutual recursion

Here is an example of a forward declaration that permits mutual recursion (Listing 8.4):

Listing 8.4 Example FORWARD Declaration With Recursion

```
{ Forward declaration }
PROCEDURE walter(m, n: INTEGER); FORWARD;

PROCEDURE clara(x, y: REAL);
  BEGIN
    { OK because walter is forward declared. }
    walter(4, 5);
  END;

PROCEDURE walter;{ Defining declaration }
  BEGIN
   clara(8.3, 2.4);
  END;
```

NOTE: Forward procedures and functions may not be included in the interface part of a unit. See "Unit Structure" on page 170 for details about the different parts of a unit, including the interface part.

The EXTERNAL and C Directives

A procedure or function declaration containing the directive `EXTERNAL` instead of a block defines the Pascal interface to a separately assembled or compiled routine, such as a procedure code module in assembly language. The external code must be linked with the compiled Pascal host program before execution; see the Linker instructions in the *CodeWarrior IDE User's Guide*.

When you use the `C` directive in addition to `EXTERNAL`, the parameters (and function return values) are automatically arranged according to C language standards. As with other external routines, C-declared procedures and functions have no body; they are linked with C compiler output by the Linker.

NOTE: There are a variety of issues involved in using C procedures and functions within your Pascal code. These issues are addressed in the section entitled "Using C Routines Within Pascal" on page 146 and in the "Pascal Notes" chapter of the Pascal Compilers Reference.

Here are two examples of external procedure declarations:

```
PROCEDURE MakeScreen(index: INTEGER); EXTERNAL;
PROCEDURE Allen(howl: STRING); C; EXTERNAL;
```

In these examples, `MakeScreen` is an external procedure that must be linked to the host program before execution. `Allen` is a C function that must be linked to the host program before execution.

It is the programmer's responsibility to ensure that the external procedure or function is compatible with the `EXTERNAL` and `C` declarations in the Pascal program; the linker does not check for compatibility.

NOTE: External procedures and functions may not be included in the interface part of a unit. See "Unit Structure" on page 170 for details about the different parts of a unit, including the interface part.

External Declarations Of Objects

When declaring an object that contains a variable of another object type that must be defined in a different source code file or library, use the `EXTERNAL` directive. Objects declared as `EXTERNAL` cannot be inherited from and their fields and methods cannot be referenced.

Declare an object as `EXTERNAL` to avoid circular references for interfaces to object declarations.

For information on circular references, see "Circular References" on page 177.

For information on objects, see "Object Pascal Overview" on page 147.

The ASM directive

The `ASM` directive is used in a routine definition to declare that it contains assembly language.

For more information on using assembly language from within Pascal see the "Assembler" chapters in the *Pascal Compilers Reference*.

Parameters

Procedure and function declarations may have any or all of these kinds of formal parameters:

- Value Parameters—a parameter group without a preceding `VAR` is a list of value parameters
- Variable Parameters—a parameter group preceded by `VAR` is a list of variable parameters

- Constant Parameters—a parameter group with a preceding `CONST` is a list of constant parameters
- Typeless Parameters—a parameter group without type specification
- Procedural Parameters—a procedure heading denotes a procedural parameter
- Functional Parameters—a function heading denotes a functional parameter
- Open Area Parameters—an unbounded array type denotes and open area parameter
- UNIV Parameters—a parameter group with a preceding `UNIV` is a list of constant parameters
- The Ellipsis Parameter (...)—an ellipsis can be used in certain cases as the last parameter in a parameter list

A formal parameter list may be part of a procedure declaration or function declaration, or it may be part of the declaration of a procedural or functional parameter. The diagram below describe the actual syntax of a formal parameter list:

<formal parameter list> ::=

((<parameter declaration> | <procedure heading> | <function heading>) { **;** (<parameter declaration> | <procedure heading> | <function heading>) } **)**

<parameter declaration> ::=

[**VAR** | **CONST**] <identifier list> **:** [**UNIV**] (<type identifier> | **ARRAY** [**OF** <type identifier>])) | (**VAR** <identifier list>)

NOTE: The types of formal parameters are denoted by type identifiers, so you cannot define a new type in a parameter list. In other words, only a simple identifier can be used to denote a type in a formal parameter list. To use a type such as `PACKED ARRAY[0..255] OF CHAR` as the type of a parameter, you must first declare a type identifier for this type:

```
TYPE ChArray = PACKED ARRAY[0..255] OF CHAR;
```

The identifier `ChArray` can then be used in a formal parameter list to denote the type.

If a formal parameter list is part of a procedure declaration or function declaration, it declares the formal parameters of the procedure or function. Each parameter so declared is local to the procedure or function being declared and can be referred to by its identifier in the block associated with the procedure or function.

If the list is part of the declaration of a procedural or functional parameter, it declares the formal parameters of the procedural or functional parameter. In this case, there is no associated block and the identifiers of parameters in the formal parameter list are significant only to the extent that they indicate the format and number of parameters.

NOTE: The word `FILE` (for an untyped file) is not allowed as a type identifier in a parameter declaration, because it is a reserved word. To use a parameter of this type, declare some other identifier for the type `FILE`. For example,

```
TYPE phyle = FILE;
```

The identifier `phyle` can then be used in a formal parameter list to denote the type `FILE`.

Value Parameters

A formal value parameter acts like a variable local to the procedure or function, except that it gets its initial value from its corresponding parameter in the parameter list.

No changes made to a formal value parameter change the value of its corresponding item in the parameter list.

For a value parameter, the corresponding actual parameter in a procedure statement or function call must be an expression, and its value must not be of a file type or of any structured type that contains a file type.

The actual parameter must be assignment-compatible with the type of the formal value parameter. However, you can override this restriction by declaring the parameter as `UNIV` ("UNIV Parameters" on page 142).

For details on *assignment compatibility*, see "Assignment Compatibility" on page 68.

Variable Parameters

Variable parameters are used when a value must be passed back from a procedure or function to the calling program.

The corresponding actual parameter in a procedure statement or function call must be a variable access, as defined in the section "Variable Accesses" on page 80. The formal variable parameter denotes the actual variable during the entire activation of the procedure or function, so any changes to the value of the formal variable parameter are reflected in the actual parameter.

Within the procedure or function, any access of the formal variable parameter is an access of the actual parameter itself. The type of the actual parameter must be identical to that of the formal variable parameter. However, you can override this restriction by declaring the parameter as `UNIV`, as described later in this chapter.

File types must be passed as variable parameters.

Only byte-aligned fields of packed structures are valid as variable parameters. To find out more about alignment of packed structures, see the platform "Notes" chapters in the *Pascal Compilers Reference*.

Constant Parameters

Constant parameters are parameters that the procedure or function cannot modify.

Within the procedure or function, any assignment to the variable corresponding to the constant parameter will give a compiler error. Also, passing the variable to a procedure or function as a variable parameter will give a compiler error.

Listing 8.5 Constant parameter example

```
TYPE
  MyRecType = RECORD
                a : LONGINT;
                c : CHAR;
              END;

PROCEDURE VarProc(VAR i : LONGINT);
BEGIN
(* ... *)
END;

PROCEDURE VarProc(i : LONGINT);
BEGIN
(* ... *)
END;

PROCEDURE ConstProc(CONST f : MyRecType);
VAR
  i : LONGINT;
BEGIN
  f.a := 1;          (* Error *)
  i := f.a;          (* Allowed *)
  VarProc(f.a);      (* Error *)
  ValProc(f.a);      (* Allowed *)
END;
```

Typeless Parameters

Typeless parameters are passed to a routine as variable parameters, but without any type specification. A routine that receives a typeless parameter must typecast it when accessing and manipulating the parameter.

Listing 8.6 shows an example of defining and calling a procedure with typeless parameters.

A program may use typeless parameters only if the **Turbo Pascal I/O Model** is selected in the **Pascal Language Settings** panel. See the Pascal Compilers Reference for more information on compiler settings.

Listing 8.6 Using typeless parameters

```
FUNCTION Equal(kind : MyDataType; VAR x, y) : BOOLEAN ;
BEGIN
  CASE kind OF
    ISCHAR:
      compare := CHAR(x) = CHAR(y);
    ISINTEGER:
      compare := INTEGER(x) = INTEGER(y);
    ISUSER:
      compare := UserDefinedType(x) = UserDefinedType(y);
  END;
END;
```

Procedural Parameters

When the formal parameter is a procedure heading, the corresponding actual parameter in a procedure statement or function call must be a procedure identifier. The identifier in the formal procedure heading represents the actual procedure during execution of the procedure or function receiving the procedural parameter.

Here are some examples of procedural parameters (Listing 8.7):

Listing 8.7 Example Procedural Parameters

```
PROGRAM passProc;
  VAR i: INTEGER;

  {x is a formal procedural}
  PROCEDURE a(x : PROCEDURE);
    BEGIN  {parameter.}
      Write('About to call x ');
      x; {Call the PROCEDURE passed as}
    END;  {parameter.}

  PROCEDURE b;
    BEGIN
      Write('In PROCEDURE b')
    END;

  FUNCTION c(x : PROCEDURE): INTEGER;
    BEGIN
      x;{Call the PROCEDURE x, passed as}
      c:=2  {formal procedural parameter.}
    END;

  BEGIN
    a(b);{Call a, passing b as parameter.}
    i:= c(b){Call c, passing b as parameter.}
  END.
```

If the actual procedure and the formal procedure have formal parameter lists, the formal parameter lists must be compatible, as described in "Parameter List Compatibility" on page 143. However, only the identifier of the actual procedure is written as an actual parameter; no parameters are given for the actual procedure.

Here is an example of procedural parameters with their own formal parameter lists (Listing 8.8):

Listing 8.8 Procedural Parameters with Formal Parameters

```
PROGRAM test;
  PROCEDURE xAsPar(y: INTEGER);
    BEGIN
      writeln('y=', y)
    END;
  PROCEDURE callProc(PROCEDURE xAgain(z: INTEGER));
    BEGIN
      xAgain(1)
    END;
  BEGIN     {body of program}
    callProc(xAsPar)

    {Note only the PROCEDURE identifier is given.}
  END.
```

If the procedural parameter, upon activation, accesses any non-local entity (by variable access, procedure statement, function call, or label), the entity accessed must be one that was accessible to the procedure when the procedure was passed as an actual parameter. To see what this means, consider a procedure `Proc` that is local to another procedure, `firstPasser`.

Suppose that the following sequence takes place:

1. **`FirstPasser` is executing.**
2. **`FirstPasser` calls a procedure named `firstReceiver`, passing `Proc` as an actual parameter.**
3. **`FirstReceiver` calls `secondReceiver`, again passing `Proc` as an actual parameter.**
4. **`SecondReceiver` calls `Proc` (first execution of `Proc`).**
5. **`SecondReceiver` calls `thirdReceiver`, again passing `Proc` as an actual parameter.**
6. **`ThirdReceiver` calls `firstPasser` (indirect recursion) and passes `Proc` to `firstPasser` as an actual parameter.**
7. **`FirstPasser` (executing recursively) calls `Proc` (second execution of `Proc`).**

Thus the procedure `Proc` is called first from `secondReceiver` and then from the second (recursive) execution of `firstPasser`.

Suppose that `Proc` uses a variable access `pVar` and `pVar` is not local to `Proc`, and suppose that each of the other procedures has a local variable named `pVar`.

Each time `Proc` is called, which `pVar` does it access? The answer is that in each case, `Proc` accesses the `pVar` that is local to the first execution of `firstPasser`—that is, the `pVar` that was accessible when `Proc` was originally passed as an actual parameter.

Procedure pointers

The `@` operator can create procedure pointers. See "The @ Operator" on page 101 for details on procedure pointers.

Functional Parameters

When the formal parameter is a function heading, the actual parameter must be a function identifier. The identifier in the formal function heading represents the actual function during the execution of the procedure or function receiving the functional parameter.

Functional parameters are exactly like procedural parameters, with the additional rule that corresponding formal and actual functions must have identical result types. See the above section "Procedural Parameters" on page 138 for more details.

Open Area Parameters

Boundless array parameters can be declared as `CONST`, `VAR` or value parameters.

The declaration is

```
param : ARRAY OF sometype
```

These parameters are accessed as `arrays [0..HIGH(param)] of` some type. The standard function `HIGH` returns the propound starting from 0. Type compatibility requires same element type because there can't be type conversion or individual elements of an argu-

ment array. Note that a string is not compatible with an `ARRAY OF CHAR`.

Here is an example of an Open Area Parameter (Listing 8.9):

Listing 8.9 Example Open Area Parameter Use

```
Procedure process(a: ARRAY OF INTEGER)
VAR i: integer;
BEGIN
    FOR i := 0 TO HIGH(a) DO
        WriteLn(a[i]);
END;

VAR i: integer;
    vi: ARRAY [1..10] OF INTEGER;

BEGIN
    process(i);
    process(vi);
END.
```

UNIV Parameters

When the word `UNIV` is given before the type identifier in the formal parameter list, the corresponding item in an actual parameter list can be of any type that is the same size as the formal parameter's type.

A discussion of built-in type data sizes can be found in the platform "Notes" chapters in the *Pascal Compilers Reference*.

Here is an example of a `UNIV` parameter (Listing 8.10):

Listing 8.10 Example UNIV Parameter Use

```
TYPE
  ptr1 = ^CHAR;
  ptr2 = ^INTEGER;
```

```
VAR
  four: LONGINT;
  pInt: ptr2;

PROCEDURE RealAddr(virt: LONGINT; rAddr: UNIV ptr1);
---
RealAddr(v, pInt);{pInt can be a pointer to a type}
                  {other than char, or can be}
RealAddr(v, four); {any other four-byte type.}
```

The Ellipsis Parameter (...)

The compiler supports the ellipsis as a parameter if it is the last parameter and the routine is declared as `C` and `EXTERNAL`.

The ellipsis is typed as three consecutive period characters (`.`) regardless of whether or not the character set you are using contains a special ellipsis character (as the Macintosh extended character set does.)

Listing 8.11 Example of the ellipsis parameter

```
PROCEDURE printf(str:STRING; ... ); C; EXTERNAL;
```

See "The EXTERNAL and C Directives" on page 132 for more information.

Parameter List Compatibility

Parameter list compatibility is required of the parameter lists of corresponding formal and actual procedural or functional parameters.

Two formal parameter lists are compatible if they contain the same number of parameters and if the parameters in corresponding positions match. Two parameters match if one of the following is true:

- They are both value parameters of identical type.
- They are both variable parameters of identical type.

- The formal parameter has `UNIV` before its type, and the actual parameter is a value or variable of the same size. The parameters must still be both value parameters or both variable parameters.
- A `PACKED ARRAY OF CHAR` is compatible with a `STRING` formal parameter if it is passed by value. In this case, the `PACKED ARRAY OF CHAR` is converted to an item of type `STRING`.
- They are both procedural parameters with compatible parameter lists.
- They are both functional parameters with compatible parameter lists and identical result types.

See "Type Compatibility" on page 66 for more information.

Invoking Procedures and Functions

Invoking a routine (a procedure or a function) refers to the execution of code within one block *jumping* into the code of the routine block, executing the routine, and then jumping back into the same place in the calling block. Once defined, procedures and functions can be invoked from anywhere within the scope of their declarations.

For information about the scope of a declared identifier, see "Scope Rules" on page 164.

Procedures and Functions are invoked differently. The following sections outline how each is invoked from within a block of Pascal code:

- Invoking Procedures
- Invoking Functions

Invoking Procedures

A procedure is activated by a procedure statement, as defined in "The Procedure Statement" on page 112, which gives the procedure's identifier and any actual parameters required by the procedure. The statements to be executed upon activation of the proce-

dure are specified by the statement part of the procedure's block. If the procedure's identifier is used in a procedure statement within the procedure's block, the procedure is executed recursively.

All actual parameters passed into the procedure must match the formal parameters as defined in the procedure declaration. For details about compatibility between actual and formal parameters, see "Parameter List Compatibility" on page 143.

The procedure statement has the following syntax:

<procedure statement> ::=

    <procedure identifier> [<actual parameter list>]

<actual parameter list> ::=

    ([<actual parameter> { **,** <actual parameter>}] **)**

Invoking Functions

A function is activated by the evaluation of a function call (see "Atomic Expressions" on page 88), which gives the function's identifier and any actual parameters required by the function. The function call usually appears as an operand in an expression. The expression is evaluated by executing the function and replacing the function call with the value returned by the function.

A function call has the following syntax:

<function call> ::=

    <function identifier> [<actual parameter list>]

<actual parameter list> ::=

    ([<actual parameter> { **,** <actual parameter>}] **)**

Function calls can also be used with variable qualifiers to identify a variable to which a value is assigned. In that case, the function is executed, the qualifiers are applied to the result of the function, and the result of the expression is assigned to the variable thus located. It is not specified whether the expression or the function is evaluated first.

If the function's identifier is used in a function call within the function's block, the function is executed recursively.

All actual parameters passed into the function must match the formal parameters as defined in the function declaration. For details about compatibility between actual and formal parameters, see "Parameter List Compatibility" on page 143.

Using C Routines Within Pascal

As described in the section entitled "The EXTERNAL and C Directives" on page 132, C procedures and functions and can be declared to be invoked from within Pascal programs. In addition to the normal complications involved in declaring an external routine (possibility of a parameter/argument inconsistency), there are a slew of other issues involved when invoking C routines from within Pascal code.

The majority of the issues involve making sure calling conventions and data types are compatible between the two languages. These issues are fundamentally related to the implementation of the CodeWarrior Pascal compiler, and as such are outside the realm of the current manual. They are, however, discussed at length in the appropriate *Targeting* manual.

9

Object Pascal

This chapter introduces the philosophy behind object-oriented programming and the mechanisms built into CodeWarrior Pascal that support it.

Object Pascal Overview

Object-oriented programming is a powerful programming methodology which revolves around the grouping of data and operations performed on that data into *objects*. CodeWarrior Pascal provides the primitives necessary to make use of such objects in your pascal programs.

The chapter starts with an introduction to the fundamental concepts and terminology behind object-oriented programming, and then goes into describing the specifics of *Object Pascal*, the set of object oriented primitives built into CodeWarrior Pascal.

NOTE: Objects in Object Pascal are not directly compatible with C++ objects. There is a section in the "Pascal Notes" chapter of the Pascal Compilers Reference which discusses how to make use of C++ in your Pascal program.

The rest of the chapter covers the following topics:

- Object Oriented Programming Basics
- Object Pascal Basics
- Creating Objects
- Declaring Methods
- Calling Methods

Object Oriented Programming Basics

This section describes some of the basic background and terminology behind object-oriented programming. It covers the following topics:

- Procedural Programming and Object Oriented Programming
- Definitions

Procedural Programming and Object Oriented Programming

Procedural programming methods, or traditional programming, uses a top-down technique for program design: a main goal is defined then sub-goals are defined to support the main goal. In procedural programming data structures are designed for a program, then routines are written to operate on those data structures.

Maintaining and improving a procedural program can be difficult. The dependencies between data and the routines that operate on that data aren't always clear. Changing a data structure may unintentionally affect other data structures and routines.

Object oriented programming (OOP) encapsulates data and routines into discrete *objects*. Objects interact with each other through defined interfaces. An object's interface is independent of its implementation. In other words, changing an object internally will not affect its use externally. Also, objects can *inherit* properties of other objects and change or add to them. Inheriting from a tested, debugged object means that bugs are probably to be found in new properties.

OOP, then, is powerful and flexible because it allows objects to improve on other objects through inheritance, and it allows defined, implementation-independent interfaces to the outside. The end goal is that changes and fixes performed on an OO program are simpler because the scope of the problem is (in theory) centralized; it is limited to some object or other rather than being distributed around various distant portions of a program.

Definitions

An *object* provides a way to join data and routines that operate on that data together. An object defines how data can be accessed and used while allowing its implementation to be hidden.

A *data member* is a field in an object.

A *method* is a routine defined as part of an object. Typically, a method only operates on the data within the object of its type.

An object is called a *child* or *descendant* when it inherits from another object. The object being inherited from is called an *ancestor* or *parent*. A child object inherits all its parent's data and methods by default. A child can override its default data and methods to change the functionality of its parent. Additionally, it can have its own, new methods.

An *instance* is simply a variable that has an object type. The distinction between an object and the instance of an object is important. An object is simply a special kind of data type that includes routines in its definitions. An instance of an object is a variable declared with that object type.

Object Pascal Basics

NOTE: In order to use Object Pascal extensions within a CodeWarrior Pascal project, there is a compiler option that must be set. See the "Pascal Settings" chapter in the Pascal Compilers Reference for details about the "Object Pascal" setting.

Objects in Object Pascal are closely related to records. Like a record, an object consists of a number of fields, each of which may be of a different type. Objects add an extra "dimension" to the idea of a record—they include not only data fields but also private procedures and functions (methods). A method that is declared in an object type definition typically operates on the data stored in an object

of that type, although it is free to refer to any other variables in its scope.

Here is an example of three object type declarations (Listing 9.1). Notice the use of a unit to implement these Pascal objects. A unit's interface and implementation parts complement an object type's declaration part and its method definition part smoothly.

Listing 9.1 Example object type declarations

```
UNIT PayRoll;

INTERFACE

TYPE

Employee = OBJECT
  firstName, lastName : STRING[32];
  hourlyWage : INTEGER;

  PROCEDURE Hire(
                name1, name2 : STRING[32];
                rate : INTEGER);
  FUNCTION Pay(hoursWorked : INTEGER) : INTEGER;
  PROCEDURE IssuePaycheck(hoursWorked : INTEGER);
END;

ExemptEmployee = OBJECT(Employee)
  FUNCTION Pay(hoursWorked : INTEGER) : INTEGER; OVERRIDE;
END;

Executive = OBJECT(ExemptEmployee)
  weeklyBonus : INTEGER;

  PROCEDURE SetBonus(performanceLevel : INTEGER);

(* Implementation continued in Listing 9.2 *)
...
```

In this example, the object type `Employee` begins the chain of inheritance. `Employee` isn't defined in terms of any other object—it doesn't have any ancestors. Like a record, an object of type `Employee` consists of several data fields.

But in addition to data, an `Employee` object has three methods: the `Hire` and `IssuePaycheck` procedures, and the `RegularPay` function. These methods are declared as if they were `FORWARD` routines. Their definitions come later in the program.

The object type `ExemptEmployee` is a descendant of `Employee`. In it, the function `RegularPay` is changed by overriding the function's body. A method that overrides a parent method must keep its parent's parameters in its method declaration. Also, `ExemptEmployee` inherits unchanged all the data fields of `Employee`, as well as the procedures `Hire` and `IssuePaycheck`.

The object `Executive` is a descendant of `ExemptEmployee`. It retains data fields of `ExemptEmployee` (which are also fields of `Employee`) and adds the field `weeklyBonus`. It also adds a new procedure, `SetBonus`. In addition, it changes the `IssuePaycheck` by overriding it. This lets `IssuePaycheck` operate on the value of `weeklyBonus`.

It is an important feature of this kind of programming that object type declarations do not need to include declarations that occur inside their ancestor objects; these declarations are automatically present by inheritance. They need to declare only changes. For example, the object types `ExemptEmployee` and `Executive` automatically include the procedure `Hire`, because they inherited it from `Employee`. As a result, you can create a very complex object type with a few simple declarations, just by naming another object type from which it inherits its structure.

For information on methods, see "Declaring Methods" on page 156.

For information on overriding, see "The OVERRIDE directive" on page 157.

Creating Objects

This section describes how to define and create object variables. The topics in this section are:

- Declaring Object Types
- Object and Member Scope
- Object Type Membership
- Declaring Object Instances
- Object Reference Variables

Declaring Object Types

Unlike other types, an object type can only be declared in the type declaration part of a main program or unit. You cannot declare an object type in a variable declaration part or in a procedure or function declaration block.

<object type> ::=

OBJECT [**(** <object type identifier> **)**] { <object data heading> | <method heading> } **END**

<object type identifier> ::=

<identifier>

<method heading> ::=

(<function heading> | <procedure heading>) [**; OVERRIDE**] { **;** <method heading> [**; OVERRIDE**] } [**;**]

<object data heading> ::=

<object data field scope> <field declaration>**;**

{<object data field scope> <field declaration> **;**}

<object data field scope> ::=

PROTECTED | **PRIVATE** | **PUBLIC**

<field declaration> ::=

<identifier list> **:** <type>

When you use an object type in a variable declaration part, you create a reference variable for that type. The reference variable stores a handle (a pointer to a pointer) to an object of that type (or a descendant type). You always access objects through reference variables. Hence you do not create special types for object references.

Instead, an object type is used to declare each variable that can hold a reference to an object of that type (or a descendant type).

Object and Member Scope

The scope of an object's identifier and its method identifiers extends over all descendants of that type, and over procedure and function blocks that implement methods of that type, its descendants, and to the scope that an instance of the object was declared in.

On the other hand, the scope of an object's data fields can be limited using the `PROTECTED`, `PRIVATE`, and `PUBLIC` keywords. By default, the scope of an object's data fields is public, the same as its method identifiers. To modify the scope of an object's data fields, use the `PROTECTED`, `PRIVATE`, and `PUBLIC` keywords when defining an object's data fields.

The `PROTECTED` keyword specifies that a field is only accessible by the object's methods and the methods of its descendants. The `PRIVATE` keyword specifies that a field is only accessible by the object's methods. The `PUBLIC` keyword specifies that a field is accessible by the object's methods, those of its descendants, and is accessible in the scope in which the object is instantiated.

Listing 9.2 Example object method definitions

```
...
(* Continued from Listing 9.1 *)
IMPLEMENTATION

PROCEDURE Employee.Hire(name1, name2 : STRING[32];
                        rate : INTEGER);
BEGIN
```

```
  firstName := name1;
  lastName := name2;
  hourlyWage := rate;
END;

FUNCTION Employee.Pay(hoursWorked : INTEGER) : INTEGER;
BEGIN
  (* Pay with deduction for benefits *)
  Pay := hourlyWage * hoursWorked - 100;
END;

FUNCTION ExemptEmployee.Pay(hoursWorked:INTEGER) : INTEGER;
BEGIN
  (* Pay with no deductions *)
  Pay := hourlyWage * hoursWorked;
END;
...
```

For an illustration of object type declarations, see Listing 9.1.

For related information on declaring objects within objects, see "Forward declaration of objects" on page 130 and "External Declarations Of Objects" on page 133.

Object Type Membership

When an object of a specific type is created during program execution, it is considered a member of that type and of all ancestral types. In Listing 9.1, an object created as an Executive is a member of types Executive, ExemptEmployee, and Employee. References to the object of type Executive may be assigned to object reference variables of types Executive, ExemptEmployee, and Employee. Which particular version of an overridden method is executed when a method call is executed depends on the type of the object, not the type of the reference variable.

Declaring Object Instances

You do not create instances from object types in the same way you create ordinary variables from other variable types. Instead, declare an object reference variable, then use the standard New procedure to create an object of a given type and assign the value that identifies the new object to the object reference variable. The New procedure sets aside a part of dynamic memory for the object, and returns a handle for the object.

An object reference variable points to the object instance through the handle returned by `New`.

NOTE: This is an extension to the standard Pascal `New` procedure. When `New` is given an ordinary pointer variable, it reserves space for an identified variable and returns a pointer to the identified variable. When `New` is given an object type variable, it creates the object in dynamic memory and returns a handle to the object.

For an example on creating an instance of an object, see Listing 9.3.

Object Reference Variables

A variable that is declared using an object type is an object reference variable. An object reference variable is not itself an object instance. The value of an object reference variable is either NIL or a value that identifies the object instance, called the identified object of the reference. Object instances themselves are dynamic variables. An object reference variable that references an object instance does so by means of a handle—a pointer to a pointer.

The pointer symbol (`^`) used to denote the identified variable of a pointer is not allowed after an object reference variable. Similarly, the double pointer symbol (`^^`) used to de-reference a handle is not allowed. Hence there is no way to treat the identified object as a variable in its own right. However, you can access components of the object through a reference variable as you would access fields of a record.

Data is stored in fields of objects, and you access those fields by giving the reference variable identifier, a period, and the field name, which appears the same as a record field access. As with records, you can omit the reference variable identifier and period under certain circumstances. Variable accesses using the reference variable identifier alone access the pointer type value stored in the reference variable, just as with other pointers.

The object reference variable and the period (.) can be omitted inside a `WITH` statement that lists the reference variable, or within any method block that declares a method of the object's type.

Listing 9.3 is an example of how a variable declared as object type Employee can be used to refer to an object of type Executive, using the type declarations in Listing 9.1.

Listing 9.3 Declaring an instance of an object and membership

```
...
VAR
  anEmployee : Employee;
  anExecutive : Executive;

BEGIN
  New(anExecutive);
  anEmployee := anExecutive;
  anEmployee.IssuePaycheck(40);
...
```

Declaring Methods

A method is a procedure or function that is declared as part of an object type declaration. Methods are declared like other procedures or functions, except that they are always declared in the stye of a forward declaration but without the keyword `FORWARD`. Object types are often declared in the interface part of a unit, while the bodies of their methods are defined in the implementation part of the same unit.

When declaring a method, the initial heading declaration and the block declaration must both appear in the main program or must both appear in the same compilation unit. The heading declaration appears in the type declaration part, as part of the object type declaration. The declaration of the method's block appears in the procedure and function declaration part. You must write the object type and a period along with the procedure or function identifier when you declare the block. You may repeat the formal parameter list; if you do, it must be identical to the original.

The OVERRIDE directive

A descendant of an object type always inherits all fields and methods of its ancestors. It can add fields and methods to those it has inherited, and it can override the action of methods. To override a method, you follow the method heading with the keyword OVERRIDE. When a method is overridden, the action of the method is changed but the interface to the method must remain exactly the same. It must retain the same spelling of all identifiers and the same data types for the method's format parameters and return value (if any).

The self parameter

In addition to ordinary parameters, every method has a default, implicit parameter, called `Self`. `Self` is a reference to the object used to call the method. It's type is the reference type of the object type to which the method belongs. (Note that this is not necessarily the type of the actual parameter supplied by the caller. `Self` could refer to an object of a descendant type.) The scope of `Self` extends over the method declaration block. You can assign values to fields of `Self`, but you cannot change the value of `Self`, which would cause `Self` to reference another object. The method acts as if its entire statement were surrounded by

```
WITH Self DO BEGIN
...
END
```

so you do not have to give the identifier `Self` when accessing fields or methods of `Self`.

For information on object inheritance and membership, see "Object Type Membership" on page 154.

Calling Methods

A method call is a special case of a function call. The same rules apply to methods and method calls for both procedures and functions.

As with fields of objects or records, a method can be accessed by using a qualifier or by using a `WITH` statement.

Here are some examples of procedural method calls:

```
v.Draw
WITH v DO Draw
x[i].Track(y + 1, z - 1);
```

Here are some examples of functional method calls:

```
IF a.Times(b) > 10 THEN ...
y := x.Extent;
WITH x DO y := Extent;
```

Unlike a field, which can be evaluated or assigned a new value, a method is executed when it is called. When a method is called, a reference to the specific object through which the method was accessed (for example, object `v` in `v.Draw`) is bound to the automatically declared formal parameter `Self`, whose type is the object type of which the method is part.

If `y` is declared to be a variable of object type `TView`, then at any moment during execution, `y` may be a reference to an object of any type that inherits from `TView`. If `TView` has a method `Draw`, for example, different inheriting types may define different implementations of the method `Draw`. Executing `y.Draw` calls the implementation of `Draw` defined for the type of the object that `y` refers to at the time of the call. The value of `Self` in the called method becomes a reference to that same object. Note that `y` may be a variable of a different type—the method actually called is the one that belongs to the object, not the variable.

One way to read the statement

```
y.Draw
```

is "Tell `y` to draw." Reading it that way points out that the program only says the object referred to by `y` should draw. At runtime, the current definition of y determines how it should draw by choosing the appropriate implementation of `Draw` for its object type.

The INHERITED directive

When one object type inherits from another but overrides an inherited method with its own implementation, you may want to call the overridden method from within the new method.

For example, if object type `Truck` inherits from `Vehicle` and overrides method `Accelerate`, then `Truck.Accelerate` may wish to invoke `Vehicle.Accelerate` at one or more points. You can call the overridden method with the special form

```
INHERITED Accelerate
```

In general, to call a method that belongs to the immediate ancestor of the object type that owns the current method, write `INHERITED` (without a period). The value of `Self` in the called method is that same value as it is in the calling method.

`INHERITED` may only be used within a declaration block of a method that is being overridden. It must precede a method identifier that was inherited by the object type that owns the method declaration block.

10 Blocks and Scope

This chapter discusses the concept of a *block* within the Pascal language In addition, it describes the declaration scope rules as they pertain to declarations within blocks.

Blocks and Scope Overview

Executable code in a Pascal program consists of one or more *blocks*; In fact, every executable line of Pascal must fall within a block. The Pascal *program* itself always contains a single block, as do all procedures and functions used by the program (for information on programs, see "Procedures and Functions Overview" on page 125; for information on procedures and functions, see "Procedures and Functions Overview" on page 125.)

In addition to defining a chunk of executable statements, a block also contains declarations for identifiers which are *local* to the block, meaning that they will only have the declared meaning within the bounds of the block itself. This realm within which a declared identifier has meaning is called the *scope* of the declaration. Scope rules are actually slightly more subtle than described here, and these subtleties will be discussed as well.

The following topics will be covered in the rest of the chapter:

- Block Structure
- Scope Rules

Block Structure

The block is the fundamental unit of Pascal source code. Each block is part of one of the following:

- A procedure declaration

- A function declaration
- A program

Each block consists of declarations and statements, constructed according to these rules:

- No specific declaration parts are required.
- Declarations parts may appear intermixed in any order.

The following diagram specifies the overall syntax of any block:

<block> ::=

{<label declaration part> | <constant declaration part> | <type declaration part> | <variable declaration part> | <procedure & function declaration part>} <statement part>

The label declaration part

The *label declaration part* declares all labels that mark statements in the corresponding statement part. Each label must mark exactly one statement in the statement part:

<label declaration part> ::=

LABEL <label> { **,** <label> } **;**

For more information about labels, see "The GOTO Statement" on page 121.

The constant declaration part

The *constant declaration part* contains all constant declarations local to the block:

<constant declaration part> ::=

CONST <constant declaration> {<constant declaration>}

For more information about constant declarations, see "Constant Declarations" on page 74.

The type declaration part

The *type declaration part* contains all type declarations local to the block:

<type declaration part> ::=

TYPE <type declaration> {<type declaration>}

For more information about type declarations, see "Type Compatibility" on page 66.

The variable declaration part

The *variable declaration part* contains all variable declarations local to the block:

<variable declaration part> ::=

VAR <variable declaration> {<variable declaration>}

For more information about variable declarations, see "Variable Declarations" on page 76.

NOTE: At run time, all uninitialized variables declared within a particular block, except for file variables, have unspecified values each time the statement part of the block is entered. File variables are initialized to `NIL`.

The procedure and function declaration part

The *procedure and function declaration part* contains all procedure and function declarations local to the block:

<procedure & function declaration part> ::=

{<procedure declaration> | <function declaration> | <method declaration>}

For more information about procedure and function declarations, see "Declaring Procedures and Functions" on page 125.

The statement part

The *statement part* specifies the actions to be executed by the block:

<statement part> ::=

    <compound statement>

For more information about compound statements, see "The Compound Statement" on page 111.

Scope Rules

The declaration of an identifier or label gives that identifier or label *meaning* within a certain portion of your Pascal program. This portion of your program in which the identifier or label is defined is called the *scope* of the declaration.

Ordinarily, the scope of an identifier or label declared within a block extends from its declaration onward to the end of the current block, including all blocks enclosed by the current block within that area.

Except in special circumstances (see "Pointer type declarations" on page 165), usage of an identifier or label within the block but *before* the declaration is illegal.

Except in special circumstances (see "Redeclaration in an enclosed block" on page 165, "Record type declarations" on page 165, and "Special rules for object types" on page 166), an identifier or label can only be declared once within a given block. Declaring the same identifier more than once is illegal.

Exceptions to the General Scope Rules

There are several exceptions to the general scope rule concerning identifier and label declarations within a block. The following exceptions are discussed below:

- Redeclaration in an enclosed block
- Pointer type declarations

- Record type declarations
- Predefined identifiers
- Special rules for object types

Redeclaration in an enclosed block

Suppose that `Outer` is a block and `Inner` is another block that is enclosed within `Outer`. If an identifier declared in block `Outer` has a further declaration in block `Inner`, then block `Inner` and all blocks enclosed by `Inner` are excluded from the scope of the declaration in block `Outer`, and the identifier assumes the new meaning as declared in `Inner`. Furthermore, if the identifier signifies a variable, the variable refers to a new piece of data… it never retains the value of the variable declared in `Outer`.

Pointer type declarations

In a pointer type declaration, the base type can be an identifier that has not yet been declared. In this case, the identifier must be declared somewhere in the same type declaration part.

Record type declarations

Normally, an identifier can not be declared twice within the same block, but there is an exception if the identifier is declared as a field within a record type declaration.

When a record field identifier is declared within a record type, it is meaningful only in combination with a reference to a variable of that record type. Therefore, an identifier declared as a field of a record type can be redeclared providing that it is not being declared within the same level of the same record type.

Predefined identifiers

CodeWarrior Pascal provides a set of predefined constants, types, procedures, and functions. The identifiers of these objects behave as if they were declared in a “super-outermost” block enclosing the entire program; thus their scope includes the entire program.

These predefined identifiers *can* be redeclared, but it is not suggested. Predefined identifiers which have been redeclared can be accessed by using the "System" qualifier (Listing 10.1).

Listing 10.1 Accessing predeclared, redefined objects

```
FUNCTION Abs(i : INTEGER) : INTEGER;
BEGIN
...
END;

...

BEGIN
  x := Abs(20); (* Use function defined in this program *)
  y := system.Abs(20); (* Use predeclared function *)
...
END;
```

Special rules for object types

In addition to having normal identifier scope, the scope of any object type identifier, object field identifier, or method identifier extends over the following areas:

- All descendants of its type.
- All procedures and function blocks that implement methods of that object type and its descendants.

The following extra redeclaration rules apply to object types and their associated identifiers:

- If you declare the identifier `OBJECT` (which is a reserved word in Object Pascal) in a program that uses Object Pascal, the compiler will issue an error.
- Object field identifiers can be redeclared in objects that are not descendants of the original object type. However, they cannot be redeclared in any descendant of the object type where they are originally declared, even if that object is declared in different block.

- Method identifiers can be redeclared, but the parameter list and return value (if any) for the new method must be identical to those for the original method.

For more information on Object Pascal see "Object Pascal Overview" on page 147.

11

Programs and Units

This chapter explains the concepts behind the two fundamental compilation elements of Pascal, the *program* and the *unit*. Also described are the declaration scope rules that revolve around these concepts.

Programs and Units Overview

Pascal blocks are assembled into *programs* and *units*. The principal difference between the two is that a program is complete and executable; a unit resembles a program but cannot be executed by itself.

Every Pascal program contains exactly one program part, but may contain any number of unit parts or, alternatively, none at all. Both programs and units are separately compiled. Their object files are then combined by the linker to form a single executable object file.

The main purpose of a unit is to factor procedures and functions out of the main program or some another unit. There are many good reasons for using units to factor routines out in this way. Here are three of the best:

- They can modularize large programs making conceptualization and maintenance easier.
- They can allow for re-using of general code for multiple Pascal projects, since a unit can be shared by more than one program.
- Breaking a program up into units can allow for the privacy of some parts of the source text while allowing the free distribution of others.

The rest of the chapter discusses the following topics:

- Program Structure
- Unit Structure

- Declaration Scope within Programs and Units
- The USES Clause

Program Structure

A Pascal program consists of a heading, an optional `USES` clause, and a block (the `USES` clause is discussed later in this chapter.) The following diagrams describe the syntax of a program:

<program> ::=

<program heading> **;** [<USES clause> **;**] <block> **.**

<program heading> ::=

PROGRAM <identifier> [**(** <program parameters> **)**]

<program parameters> ::=

<identifier list>

The occurrence of an identifier immediately after the reserved word `PROGRAM` declares it as the program's identifier. Program parameters, as described by Jensen and Wirth and the ANS Standard, are ignored by CodeWarrior Pascal.

The bulk of the *program* is found in the *block* which appears as the last part of the program structure. Blocks are the Pascal elements which contain all executable code; they are discussed in detail in "Block Structure" on page 161.

Unit Structure

The syntax for writing a unit is shown below:

<unit> ::=

<unit heading> **;** <interface part> [<implementation part>]
(<compound statement> | **END**) **.**

<unit heading> ::=

UNIT <identifier>

The *interface part* of a unit declares constants, types, variables, procedures, and functions that are "public"—that is, available to the *host program* (which may be another unit). In other words, the scope of the public entities is the entire host program. It can access these entities just as if they had been declared in its source text.

You declare procedures and functions in the interface part by giving only the procedure or function name, parameter specifications, and function result type. In other words, you give only the part that defines how the procedure or function is called, often referred to as the routine's *interface*:

<interface part> ::=

INTERFACE [[<USES clause>] {<constant declaration part> | <type declaration part> | <variable declaration part> | <procedure & function declaration part>}]

If `INLINE` or `C` directives are used within routines in the unit, the directives must also appear in the interface to the unit.

Variables and routines that appear in the interface are global. They have scope equivalent to that of entities declared in whatever block is *hosting* the unit. Hosting a unit in this way (and thereby inheriting everything in its *interface part*), is done by way of a `USES` *clause*. Furthermore, an interface itself can inherit other interfaces with the `USES` clause. For more details about using the `USES` clause, see "The USES Clause" on page 174.

The optional *implementation part*, which follows the last declaration in the interface part, declares any constants, types, variables, procedures, or functions that are "private"—that is, not available to the host program. Private procedures and functions are declared like procedures and functions in programs, with a procedure or function heading and a body. Below is a syntax description of the implementation part of a unit:

<implementation part> ::=

IMPLEMENTATION [[<USES clause>] {<constant declaration part> | <type declaration part> | <variable declaration part> | <procedure & function declaration part>}]

For further information about declaring procedures and functions, see Chapter "Declaring Procedures and Functions" on page 125.

NOTE: The CodeWarrior Pascal compiler supports a special way of allowing a procedure or function in the implementation part to be declared as public. For details, see the "Directives, Pragmas, and Predefined Symbols" chapter in the Pascal Compiler Manual.

All public procedures, functions, and methods must be redeclared in the implementation part. Parameters and function result types can be omitted from these declarations, because they were declared in the interface part; the procedure and function blocks, omitted in the interface part, are found in the implementation part. If you repeat parameter lists and function result types, they must be identical to those in the interface part.

In effect, the procedure, function, and method declarations in the interface are like forward declarations, although the `FORWARD` directive is not used. Therefore, these procedures and functions can be defined and referred to in any sequence in the implementation.

An anonymous type may be specified in the interface part of a unit while its actual type declaration appears in the implementation part. The corresponding type declaration in the implementation needs to be a pointer to some type or an object. The content of the opaque type is unavailable to any unit except to the defining implementation.

The interface part may contain a `USES` clause; thus any unit can use another unit.

There last part of a unit is an optional compound statement section, which typically contains statements to initialize the unit. There is no predefined order that the initialization sections of units in a program are executed.

NOTE: Global labels cannot be declared in a unit.

Here is short example of a unit (Listing 11.1):

Listing 11.1 Example Unit

```
UNIT Simple;

INTERFACE {public items declared}
  CONST FirstValue = 1;
  PROCEDURE AddOne(VAR Incr: INTEGER);
  FUNCTION Add1(Incr: INTEGER): INTEGER;

IMPLEMENTATION
  PROCEDURE AddOne;{Note lack of parameters...}
    BEGIN
      Incr := Incr+1
    END;

  FUNCTION Add1; {...and lack of function result type.}
    BEGIN
      Add1 := Incr+1
    END;
  END.
```

Declaration Scope within Programs and Units

The declaration of an identifier within a program or unit gives that identifier or label *meaning* within a certain portion of your Pascal program. This portion of your program in which the identifier or label is defined is called the *scope* of the declaration. This section of the chapter will describe the scope rules that apply to declarations made within a program or unit.

Scope rules within a program

Since declarations within a program are within the program's *block*, the scope rules for a program are exactly the same for those of any other block. These rules are described in "Scope Rules" on page 164.

Scope rules within a unit

While a unit is not the same as a block, declarations within a unit follow almost exactly the same scope rules as they do in a block, described in "Scope Rules" on page 164.

The only difference between the scope rules within a unit and those within a block concerns the distinction between declarations made in the implementation part vs. the interface part of a unit. Since blocks don't have these different parts, this rule doesn't apply to blocks at all.

Identifiers declared in the interface part of a unit have a scope that extends to the end of a unit. The scope of these identifiers also extends to include any other units or programs that reference the unit in a `USES` clause.

Identifiers declared in the implementation part of a unit have a scope that extends to the end of the unit. These identifiers are hidden from any other units or programs that reference the unit in a `USES` clause.

The USES Clause

You write a `USES` clause in a program or unit to access a unit. The syntax for a USES clause is as follows:

<USES clause> ::=

`USES` <identifier list>

The `USES` clause appends the ".p" suffix (which denotes source code) to the unit name and causes the compiler to open the file. For example, the statement

```
USES QuickDraw, memTypes;
```

opens the files `QuickDraw.p` and `memTypes.p`.

If **Uses Propagation** is off, the `USES` clause identifies all units required by the program. These include both units that it uses directly and any other units that are used by those units.

If **Uses Propagation** is on, the USES clause need only identify units 1 level deep. The compiler will automatically include units referred to by subsequent units.

NOTE: For more information on setting the **Uses Propagation** and other compiler options see the Pascal Compilers Reference.

In a host program, the `USES` clause (if any) immediately follows the program heading. In a host unit, the `USES` clause (if any) immediately follows the reserved word `INTERFACE`. Only one `USES` clause may appear in any host program or unit; it declares all units used by the host program or unit.

Assume that the example unit is named `Simple`. The following is a short program that uses `Simple` (Listing 11.2).

Listing 11.2 Example of Using a Unit

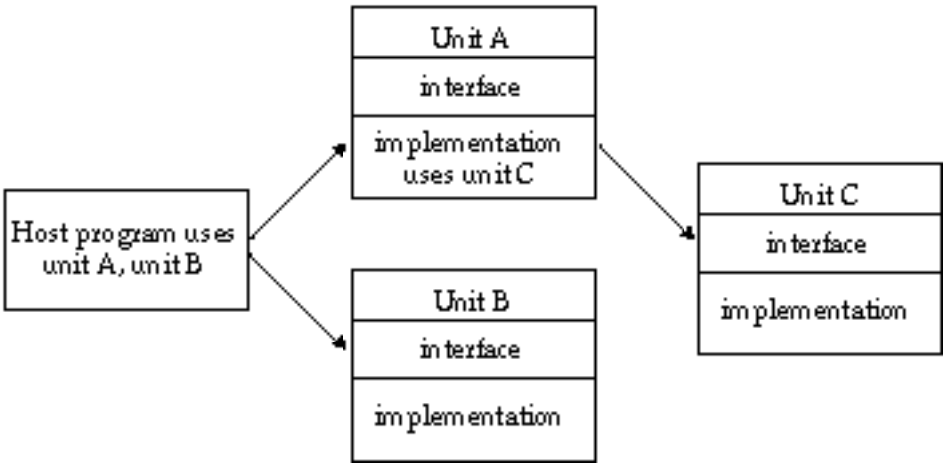
```
PROGRAM CallSimple;
  USES Simple; {use unit Simple}

  VAR i: INTEGER;

  BEGIN
    i := FirstValue;{FirstValue is from Simple.}
    write('i+1 is ', Add1(i));
    {Add1 is defined in Simple.}
END.
```

Units That Use Other Units

As explained in "The USES Clause" on page 174, the `USES` clause in the host program or unit must name all units that are required. Here "required" means that the host directly references something in the interface of the unit. Consider the unit references in Figure 11.1.

NOTE: For more information on setting the **Uses Propagation** and other compiler options see the Pascal Compilers Reference.

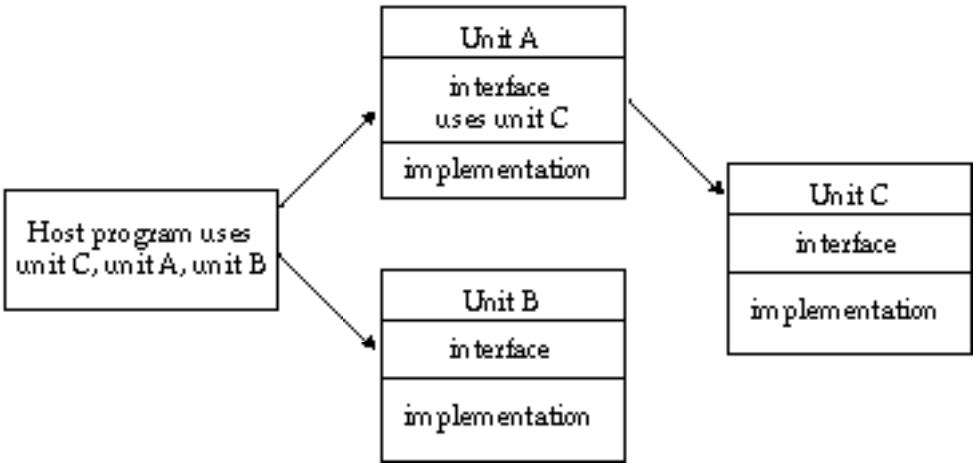
Figure 11.1 Example of simple unit references



The host program directly references the interfaces of `unitA` and `unitB`; the `USES` clause names both of these units. The implementation part of `unitA` also references the interface of `unitC`, but it is not necessary to name `unitC` in the host program's `USES` clause.

In some cases, the `USES` clause must also name a unit that is not directly referred to by the host. Figure 11.2 is exactly like Figure 11.1 except that this time the interface of `unitA` references the interface of `unitC`, and `unitC` must be named in the host program's `USES` clause. Note that `unitC` must be named before `unitA`.

Figure 11.2 Example of nested unit references



In a case like this, the documentation for `unitA` should state that `unitC` must be named in the `USES` clause before `unitA`.

Circular References

The Pascal compiler builds your application, library, or code resource by examining every Pascal file in your project file. As it examines project files, it builds a tree of dependencies for the interfaces of your units and for their implementations.

A circular reference occurs when a unit declares something that is used in another unit and that second unit declares something used by the first. To break this endless cycle of references, the Pascal compiler does not allow such things among the interface parts of units, but it is permitted for implementations.

Listing 11.3 An example of a circular reference

```
UNIT A;                 UNIT B;                 UNIT C;
INTERFACE               INTERFACE               INTERFACE
USES C;                 USES C;

TYPE                    TYPE                    TYPE
     A_type = ...            B_type = ...            C_type = ...

IMPLEMENTATION          IMPLEMENTATION          IMPLEMENTATION
USES B;                 USES A;                 USES A, B;
...                     ...                     ...
```

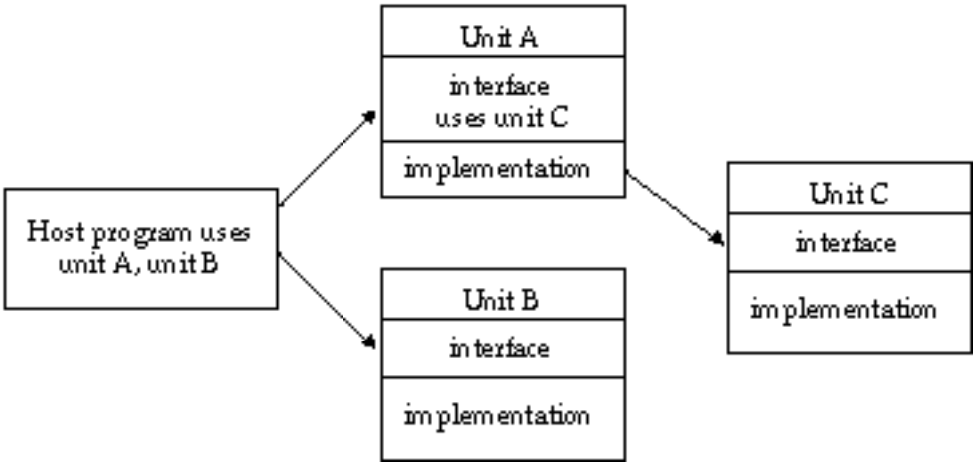
The example in Listing 11.3 is perfectly valid, since both A's and B's interfaces depend on C's, but are independent from one another. Knowing everything that was declared, A's implementation depends on all interfaces, the same is true for B's and C's.

For information on avoiding circular references for interfaces to object definitions, see "External Declarations Of Objects" on page 133.

USES propagation

The naming of units used by other units in the host program's `UNIT` clause can be difficult to manage. There is a compiler option, called "USES propagation", which makes units used by other units available to the host program (Figure 11.3).

Figure 11.3 Example of Uses Propagation



When USES propagation is enabled, the host program does not have to name unitC in its `USES` clause. The identifiers declared in unitC's interface part are available to unitA and, by propagation, available to the host program.

For information about USES propagations, see chapter "Pascal Settings" chapter of the *Pascal Compilers Reference*.

12

Files and I/O

This chapter discusses the basic file and I/O (input/output) features provided by CodeWarrior Pascal.

Files and I/O Overview

Different operating systems provide different mechanisms and levels of access for interacting with files and other I/O devices. As an alternative to interacting directly with the operating system, CodeWarrior Pascal provides its own set of mechanisms for manipulating files and I/O devices. This level of access is in a sense *abstract* from any particular platform or operating system since it can be used identically independent of which kind of target the code is being compiled for.

The CodeWarrior Pascal supported mechanisms for file and I/O manipulation can be seen as residing in two separate places. Some aspects, like the notion of a FILE-typed variable, are a core part of the language itself. Other aspects, including most of the functions used to actually deal with FILE-types variables, are not a part of the language, but rather are included with the standard Pascal libraries.

The current chapter will be concerned only with those aspects of file and I/O manipulation which are actually part of the language itself. All other aspects receive complete coverage in the "Input and Output" chapter of the *Pascal Library Reference*.

The topics discussed in the rest of the chapter are:

- Pascal File Variables
- Using Pascal Files

Pascal File Variables

A Pascal file variable is a structured variable. A file variable resembles an array, in that it consists of a sequence of distinct variable components all of the same type. However, the number of components is indeterminate, and they are not accessed by indexing but by using the predefined I/O procedures and functions outlined in the "Input and Output" chapter of the *Pascal Library Reference*.

Unlike other kinds of variables, file variables are used to store data outside of memory, in an *external file*. An external file is either a peripheral device or a named disk file. In order for a program to read or write information using an external file, a file variable must be not only be declared like any other variable, but it must also be explicitly associated with the external file itself.

This rest of this section covers the following topics concerning file variables:

- File Variable Types
- Predeclared File Variables
- The File Window Variable

File Variable Types

A file variable is declared along with other variables, using a file type. There are three basic Pascal file types:

- Structured Files
- Text Files
- Untyped Files

The syntax for writing file types is given under "Structured Types" on page 55.

Structured Files

A structured file is made up of components called *logical records*. These components may be of any type that is not a file type (nor a

structured type that contains a file type component at any level of structuring). They do not need to be of type RECORD.

For example, the declarations in Listing 12.1 create three file variables:

- IntVals is a file variable whose logical records are integer values.
- RealVals is a file variable whose logical records are real values.
- CompVals is a file variable whose logical records are of RECORD type, with an integer value and a real value in each record.

Listing 12.1 Example of Three Kinds of File Variables

```
VAR
  IntVals: FILE OF INTEGER;
  RealVals: FILE OF REAL;
  CompVals: FILE OF RECORD
        i:  INTEGER;
        r:  REAL
      END;
```

Text Files

The Pascal predefined file type TEXT can be used to store any type of data, as long as it is in character format. For example, this declaration creates a file variable of type TEXT:

```
VAR NameFile: TEXT;
```

This type of file is most efficient in handling lines of text. It transfers data in blocks between the external device and a buffer, from which your program can access the data efficiently one line or one character or value at a time.

NOTE: There are many cases where this isn't the most efficient way to perform I/O. For example, if you want to store floating-point values in a file, using a file of type TEXT would require converting

each value to its equivalent in ASCII characters before it is stored in the file. Each time a value is read from the file, it would have to be converted back to its binary representation.

Untyped Files

To declare a file variable as untyped, use the type identifier `FILE` alone. For example,

```
VAR BlockFile: FILE;
```

Pascal transfers data in and out of such a file without interpreting its internal structure. An untyped file has no file window variable, and it can only make use of a subset of the functions which a typed file can use. See the "Input and Output" chapter of the *Pascal Library Reference* for details on all supported file operations.

Predeclared File Variables

CodeWarrior Pascal sees all peripheral devices, such as the keyboard and the Macintosh screen, as external files. It predeclares two corresponding file variables of type `TEXT`, called `input` and `output`. Unless specifically redirected, `input` comes from the *standard input* and `output` goes to the *standard output*. The actual incarnations of these abstract concepts is dependent on the platform and libraries being used. See the "Input and Output" chapter of the *Pascal Library Reference* for details.

At the start of each program execution, the files `input` and `output` are automatically opened for use without being declared. Procedures and functions that can be used with files of type `TEXT` can use them, directly (receiving them as parameters) or indirectly (as the default when the file variable parameter is omitted), without declaring them first. The program should not try to close these files.

NOTE: The ANS Standard specifies that `input` and `output` must appear in the program heading if they are used. Although not required by CodeWarrior Pascal, many versions of Pascal rely on this specification. If you are writing code that may be transported

to other versions of Pascal, include `input` and `output` in your program heading.

Several of the predefined procedures and functions for use with files of type `TEXT`, described in the *Pascal Library Reference*, need not have a file variable explicitly given as a parameter. In these cases, `input` and `output` will be assumed by default, depending on whether the procedure or function is input-oriented or output-oriented.

The File Window Variable

Although a typed file variable may have any number of logical records, only one is accessible at any one time. Each logical record has a number that is its position in the file relative to the first record in the file, which is record number 0. The position of the current record in the file is called the *current file position*. Program access to the current record uses a special variable associated with the file, called a *file window variable*.

The file window variable is accessed by applying a special qualifier to the file variable. This qualifier, which immediately follows the variable, is called the *file buffer symbol*:

<file buffer symbol> ::=

```
^
```

Thus the file window variable of file `F` is referred to by `F^`.

The file window variable cannot be used with untyped files.

NOTE: Under certain conditions, such as when the current file position is at the end of the file, the value of the file variable `f` is said to be undefined. It is an error to attempt to use the value of the file window variable `f^` when the value of `f` is undefined. However, assignment to `f^` is still possible if the file may be written to.

Using Pascal Files

While most of the actual details involving file and file variable manipulations are fundamentally within the scope of the *Pascal Library Reference*, there are several important usage issues relevant to this chapter. They following topics are discussed in this section:

- Opening and Closing Files
- Sequential Versus Random Access

Opening and Closing Files

Unlike other variables, file variables can not be operated on immediately after being declared. Before actually using a file variable, the file must first be *opened*. Opening a file serves several purposes. First of all, it associates the external file with the Pascal file variable. It also alerts the operating system that your program is using the file and therefore limits the access that other running programs have to the usage of the same file.

Since other running applications have restricted access to a file while it is opened, the file must be explicitly *closed* when your program wants to give up control and allow other applications unrestricted access to the file.

There are a number of other issues involved in opening and closing files, such as different levels of access (such as read-only or read-write access) and different types of access (such as sequential or random access.) While sequential and random access are discussed briefly below, these issues are discussed in full detail in the *Pascal Library Reference*, along with exact specifications of all the functions available to perform file opening and closing operations.

Sequential Versus Random Access

Files may be accessed sequentially or randomly. When a file is opened and accessed sequentially, the first logical record is read or written and then the current file position moves to the numerically next logical record in the file. When a file is opened in random ac-

cess mode, it is possible to skip around from record to record in whatever order is desired.

NOTE: The terms random file and sequential file are commonly used but misleading. Random and sequential are two methods for accessing files—not two kinds of files. Any file can be accessed randomly or sequentially, or both ways. The predefined procedures that support random access are generally used with non-text files, but are not restricted to them.

The issue of sequential vs. random access of files is discussed in much greater detail in the "Input and Output" chapter of the *Pascal Library Reference*.

13

Syntax Summary

This appendix is a complete listing, in alphabetical order, of the syntactic elements presented throughout the book. Together as a group, this set of syntax constructions makes up a definitive specification of the CodeWarrior Pascal language.

Syntax Summary Overview

The notation used in our diagrams is called "**E**xtended **B**ackus-**N**aur **F**orm", or EBNF for short, and is a general-purpose language used for specifying computer languages. A full description of EBNF notation is can be found under "Syntax Diagrams" on page 17.

Metrowerks Pascal Syntax Specification

actual parameter

<actual parameter> ::=

<expression> | <variable access> | <procedure identifier> | <function identifier>

actual parameter list

<actual parameter list> ::=

([<actual parameter> { **,** <actual parameter>}] **)**

array type

<array type> ::=

ARRAY [<index type> { **,** <index type> } **]** **OF** <type>

assignment statement

<assignment statement> ::=

(<variable access> | <function identifier>) **:=** <expression>

atomic expression

<atomic expression> ::=

<variable access> | <unsigned constant> | <function call>

base type

<base type> ::=

<type identifier>

block

<block> ::=

{<label declaration part> | <constant declaration part> | <type declaration part> | <variable declaration part> | <procedure & function declaration part>} <statement part>

case

<case> ::=

<constant range> { **,** <constant range> } **:** <statement>

CASE statement

<CASE statement> ::=

CASE <expression> **OF** <case> { **;** <case>} [<otherwise clause>] [**;**] **END**

compound statement

<compound statement> ::=

BEGIN <statement> {**;** <statement>} **END**

constant

<constant> ::=

<constant expression>

constant atomic expression

<constant atomic expression> ::=

<unsigned constant> | <function call>

constant declaration

<constant declaration> ::=

<constant identifier> ((**=** <constant>) | (**:** <type> **=** (<constant> | <variable initialization>))) **;**

constant declaration part

<constant declaration part> ::=

CONST <constant declaration> {<constant declaration>}

constant expression

<constant expression> ::=

<simple constant expression> [(**=** | **<** | **>** | **<=** | **>=** | **<>** | **IN**) <simple constant expression>]

constant factor

<constant factor> ::=

<constant atomic expression> | <set constructor> | **(** <constant expression> **)** | **NOT** <constant factor>

constant identifier

<constant identifier> ::=

<identifier>

constant range

<constant range> ::=

<constant expression> [**..** <constant expression>]

constant term

<constant term> ::=

<constant factor> { (***** | **/** | **DIV** | **MOD** | **AND** | **&** | ******)
<constant factor>}

control variable

<control variable> ::=

<variable identifier>

CYCLE statement

<CYCLE statement> ::=

CYCLE

digit

<digit> ::=

0 | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

digit sequence

<digit sequence> ::=

<digit> {<digit>}

enumerated type

<enumerated type> ::=

(<identifier list> **)**

expression

<expression> ::=

<simple expression> [(**=** | **<** | **>** | **<=** | **>=** | **<>** | **IN**) <simple expression>]

factor

<factor> ::=

([**@**] <variable access>) | <atomic expression> | <set constructor> | **(** <expression> **)** | (**NOT** <factor>)

field declaration

<field declaration> ::=

<identifier list> **:** <type>

field designator

<field designator> ::=

. <identifier>

field list

<field list> ::=

(<variant part> | (<fixed part> [**;** <variant part>])) [**;**]

file buffer symbol

<file buffer symbol> ::=

^

file type

<file type> ::=

TEXT | (**FILE** [**OF** <type>])

final value

<final value> ::=

<expression>

fixed part

<fixed part> ::=

<field declaration> { **;** <field declaration> }

FOR statement

<FOR statement> ::=

FOR <control variable> **:=** <initial value> (**TO** | **DOWNTO**) <final value> [**BY** step value] **DO** <statement>

formal parameter list

<formal parameter list> ::=

((<parameter declaration> | <procedure heading> | <function heading>) { **;** (<parameter declaration> | <procedure heading> | <function heading>) } **)**

function body

<function body> ::=

([**ASM;**] <block>) | (**FORWARD**) | ([**C;**] **EXTERNAL**) | (**INLINE** <constant expression> { **,** <constant expression> })

function call

<function call> ::=

<function identifier> [<actual parameter list>]

function declaration

<function declaration> ::=

<function heading> **;** <function body> **;**

function heading

<function heading> ::=

FUNCTION [<object type identifier> **.**] <identifier> [<formal parameter list>] **:** <type identifier>

function identifier

<function identifier> ::=

<identifier>

function type

<function type> ::=

FUNCTION <formal parameter list> **:** <type identifier>

GOTO statement

<GOTO statement> ::=

GOTO <label>

hex digit

<hex digit> ::=

0 | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **A** | **B** | **C** | **D** | **E** | **F**

hex digit sequence

<hex digit sequence> ::=

<hex digit> {<hex digit>}

identifier

<identifier> ::=

(<letter> | <underscore>) {<letter> | <underscore> | <digit>}

identifier list

<identifier list> ::=

<identifier> { **,** <identifier>}

IF statement

<IF statement> ::=

IF <expression> **THEN** <statement> [**ELSE** <statement>]

implementation part

<implementation part> ::=

IMPLEMENTATION [[<USES clause>] {<constant declaration part> | <type declaration part> | <variable declaration part> | <procedure & function declaration part>}]

index

<index> ::=

[<expression> { **,** <expression> } **]**

index type

<index type> ::=

<ordinal type identifier>

initial value

<initial value> ::=

<expression>

interface part

<interface part> ::=

INTERFACE [[<USES clause>] {<constant declaration part> | <interface type declaration part> | <variable declaration part> | <procedure & function declaration part>}]

interface type declaration

<interface type declaration> ::=

(<identifier> **=** <type>) | (<identifier>) **;**

interface type declaration part

<interface type declaration part> ::=

TYPE <interface type declaration> {<interface type declaration>}

label

<label> ::=

<identifier>

label declaration part

<label declaration part> ::=

LABEL <label> { **,** <label> } **;**

LEAVE statement

<LEAVE statement> ::=

LEAVE

member group

<member group> ::=

<expression> [**..** <expression>]

method declaration

<method declaration> ::=

<function declaration> | <procedure declaration>

method heading

<method heading> ::=

(<function heading> | <procedure heading>) [**; OVERRIDE**] { **;** <method heading> [**; OVERRIDE**] } [**;**]

null statement

<null statement> ::=

;

object data heading

<object data heading> ::=

<object data field scope> <field declaration>**;**

{<object data field scope> <field declaration> **;**}

object data field scope

<object data field scope> ::=

PROTECTED | **PRIVATE** | **PUBLIC**

object type

<object type> ::=

OBJECT [**(** <object type identifier> **)**] { <object data heading> | <method heading> } **END**

object type identifier

<object type identifier> ::=

<identifier>

opaque type

<opaque type> ::=

<object type identifier>

opaque type identifier

<opaque type identifier> ::=

<identifier>

ordinal type identifier

<ordinal type identifier> ::=

<identifier>

otherwise clause

<otherwise clause> ::=

; OTHERWISE <statement>

parameter declaration

<parameter declaration> ::=

([**VAR** | **CONST**] <identifier list> **:** [**UNIV**] (<type identifier> | **ARRAY** [**OF** <type identifier>])) | (**VAR** <identifier list>)

NOTE: See "Typeless Parameters" on page 138 for more information on declaring parameters without type.

pointer symbol

<pointer symbol> ::=

^

pointer type

<pointer type> ::=

(<pointer symbol> <base type>) | <pointer type identifier>

pointer type identifier

<pointer type identifier> ::=

<identifier>

procedure & function declaration part

<procedure & function declaration part> ::=

{<procedure declaration> | <function declaration> | <method declaration>}

procedure body

<procedure body> ::=

([**ASM;**] <block>) | (**FORWARD**) | ([**C;**] **EXTERNAL**) | (**INLINE** <constant expression> { **,** <constant expression>})

procedure declaration

<procedure declaration> ::=

<procedure heading> **;** <procedure body> **;**

procedure heading

<procedure heading> ::=

PROCEDURE [<object type identifier> **.**] <identifier> [<formal parameter list>]

procedure identifier

<procedure identifier> ::=

<identifier>

procedure statement

<procedure statement> ::=

<procedure identifier> [<actual parameter list>]

procedure type

<procedure type> ::=

PROCEDURE <formal parameter list>

program

<program> ::=

<program heading> **;** [<USES clause> **;**] <block> **.**

program heading

<program heading> ::=

PROGRAM <identifier> [**(** <program parameters> **)**]

program parameters

<program parameters> ::=

<identifier list>

qualifier

<qualifier> ::=

<index> | <field designator> | <file buffer symbol> | <pointer symbol>

qualifier identification

<qualifier identification> ::=

<identifier> { . <identifier> }

quoted character constant

<quoted character constant> ::=

' <string character> '

quoted string constant

<quoted string constant> ::=

' {<string character>} '

real type

<real type> ::=

<real type identifier>

real type identifier

<real type identifier> ::=

<identifier>

real-type numbers

<real-type numbers> ::=

[<sign>] (**INF** | (**NaN** [**(** <digit sequence> **)**]) | (<digit sequence> [**.** [<digit sequence>]] [**E** [<sign>] <digit sequence>]))

record type

<record type> ::=

RECORD [<field list>] **END**

REPEAT statement

<REPEAT statement> ::=

REPEAT <statement> { **;** <statement> } **UNTIL** <expression>

RETURN statement

<RETURN statement> ::=

RETURN <expression>

scalar type

<scalar type> ::=

<subrange type> | <enumerated type> | <ordinal type identifier>

schema declaration

schema declaration ::=

<schema definition> **=** <array type> | <record type>

schema definition

<schema definition> ::=

<identifier> **(** <identifier list> **:** <type> { **;** <identifier list> **:** <type>} **)**

schema type

schema type ::=

<identifier> **(** <expression> { **,** <expression> } **)**

set constructor

<set constructor> ::=

[[<member group> { **,** <member group> }] **]**

set type

<set type> ::=

SET OF <scalar type>

sign

<sign> ::=

+ | **–**

simple constant expression

<simple constant expression> ::=

[<sign>] <constant term> { (**+** | **–** | **OR** | **|**) <constant term> }

simple expression

<simple expression> ::=

[<sign>] <term> { (**+** | **–** | **OR** | **|**) <term> }

simple type

<simple type> ::=

<scalar type> | <real type> | <string type>

size attribute

<size attribute> ::=

<constant expression>

statement

<statement> ::=

<assignment statement> | <compound statement> | <procedure statement> | <FOR statement> | <WHILE statement> | <REPEAT statement> | <IF statement> | <CASE statement> | <GOTO statement> | <CYCLE statement> | <LEAVE statement> | <WITH statement> | <RETURN statement> | <null statement>

statement part

<statement part> ::=

<compound statement>

step value

<step value> ::=

<constant>

string character

<string character> ::=

(**''**) | (any character except **'** or **Return**)

string type

<string type> ::=

(**STRING** [**[** <size attribute> **]**]) | (<string type identifier>)

string type identifier

<string type identifier> ::=

<identifier>

structured type

<structured type> ::=

([**PACKED**] (<array type> | <set type> | <file type> | <record type>)) | <stuctured type identifier> | <object type identifier>

stuctured type identifier

<structured type identifier> ::=

<identifier>

subrange type

<subrange type> ::=

<constant expression> **..** <constant expression>

tag field type

<tag field type> ::=

<ordinal type identifier>

term

<term> ::=

<factor> { (***** | **/** | **DIV** | **MOD** | **AND** | **&** | ******) <factor> }

type

<type> ::=

<simple type> | <structured type> | <pointer type> | <object type> | <procedure type> | <function type> | <schema type> | <opaque type>

type declaration

<type declaration> ::=

(<type identifier> **=** <type>) | <schema declaration> **;**

type declaration part

<type declaration part> ::=

TYPE <type declaration> {<type declaration>}

type identifier

<type identifier> ::=

<identifier>

underscore

<underscore> ::=

_

unit

<unit> ::=

<unit heading> **;** <interface part> [<implementation part>]
(<compound statement> | **END**) **.**

unit heading

<unit heading> ::=

UNIT <identifier>

unsigned constant

<unsigned constant> ::=

<unsigned number> | <quoted string constant> | <constant identifier> | **NIL**

unsigned number

<unsigned number> ::=

<digit sequence> | (**$** <hex digit sequence>)

USES clause

<USES clause> ::=

USES <identifier list>

variable access

<variable access> ::=

(<variable identifier> | <function call>) {<qualifier>}

variable declaration

<variable declaration> ::=

<identifier list> **:** <type> **;** [((**EXTERNAL**) | (**FAR**)) **;**]

variable declaration part

<variable declaration part> ::=

VAR <variable declaration> {<variable declaration>}

variable identifier

<variable identifier> ::=

<identifier>

variable initialization

<variable initialization> ::=

((<variable initialization> | <constant>) [**,** <variable initialization>] **)**

variant

<variant> ::=

<constant> { **,** <constant> } **:** **(** [<field list>] **)**

variant part

<variant part> ::=

CASE [<identifier> **:**] <tag field type> **OF** <variant> { **;** <variant> }

WHILE statement

<WHILE statement> ::=

WHILE <expression> **DO** <statement>

WITH statement

<WITH statement> ::=

WITH (<record variable access> | <object reference variable access>) { **,** (<record variable access> | <object reference variable access>) } **DO** <statement>

Index

Symbols

- 95
$ 34
& 96, 97
(92
* 98
** 25
+INF 44
. 85
... 143
@ 54, 101
∞ 44

Numerics

68000
    addressing 130

A

Accessing variables 80
Americal National Standards. See ANS.
Special symbols
    replacements for 32
anonymous 64, 172
ANS
    specification 20
Arrays
    assignment 111
    LONGINT as index type 57
    qualifying 110
    See also Types.
    type 56
ASCII 49, 100, 184
ASM 133
Assignment compatibility 68, 136

B

Bit manipulation 25
Blockwrite 62
BOOLEAN
    assignment 110
Boolean
    operators 96
BOOLEAN. See also Types.
Bytewrite 25, 184

C

C 129, 132, 171
C/C++
    character strings 37, 53
    continue statement 123
    from Object Pascal 147
    referencing object code 78
CASE 25, 107
CHAR. See Types.
Characters
    ASCII set 31
    Macintosh Extended set 31
Circular references 133, 177
Clancy, Michael 20
CodeWarrior
    User's Guide 132, 184
Coercion 69, 71, 110
Comments
    nesting 38
Comparing
    booleans 100
    numbers 99
    sets 101
    strings 100
CompDate 75
CompTime 75
Conditional statements 107
Constants
    declaration 74, 162
    hexadecimal 35
    predefined 165
Control statements 108
Cooper, Doug 20
CYCLE 25, 108, 122

D

Dec procedure 27, 45
Delimiters
    comments 38
    compiler directives 38
    definition 38

Design Patterns for Object-Oriented Software Development 20
Directives
definition 39
for procedures and functions 129
DO 117

E

Elements of Reusable Object-Oriented Software 20
Ellipsis, as parameter 143
END 111
Enumerated. See Types.
Exit procedure 25, 108
Exponents 34, 94
Expressions
Atomic 88
Constant 88
Compound 89
Constant 91
EXTERNAL 77

F

FAR 77
Fields
byte-aligned 136
definition 58
fixed 59
qualifying 84, 110
FILE
reserved word 135
text 24
File
Pascal 182
predeclared variables 184
See also Types.
sequential access 186
structured 182
text 183
type 62
untyped 184
File window variables
description 185
qualifiying 85
FOR 106, 114
FORWARD 129, 130, 171
Function
as parameter 141
declaration 127, 163
return type 25
type. See also Types.

G

Gamma, Erich 20

H

Halt procedure 108
Hexadecimal notation 34

I

Identifiers
allowable characters 33
case sensitivity 33
definition 33
examples 34
legal characters 33
length 24, 27, 33
scope 164, 173
significant characters 33
IN 99, 101
Inc procedure 45
+INF 44
-INF 44
Infinity 44
INHERITED 159
INLINE 171
Inside Macintosh
about 20
Text 101
INTEGER
See also Types.
INTERFACE 175

J

Jensen, Kathleen 20

L

Labels

declaration 162
definition 36
GOTO 121
redeclaration 165
LEAVE 25, 107, 108, 123
Ledgard, Henry 20
Length function 52
LONGINT
assignment 110
See also Types.
Loop control 107

M

Machine code 133
Macintosh
Extended character set 31
Mac OS routines 101
MOD 95
modulus 95
MPW
Pascal, extensions to 27

N

NaN 44, 100
Nested IF statements 119
New 24, 27, 54, 60, 155
NIL 76
Numbers
definition 34
hexadecimal 34
scientific notation 34

O

Objects
child definition 149
data member 149
definitions 149
FORWARD declarations 130
instantiation 149, 155
method 149
object definition 149
parent definition 149
reference variables 155
See also Types.
type 63, 149
OF 57
Oh! Pascal! 20
OOP 148
opaque 64, 172
Operators
^ 24
| 25
arithmetic 94
definition 93
precedence 90
set 98
string 97
Ord function 44
Ord4 function 47
OTHERWISE 120
OVERRIDE 157

P

PACKED 56, 101
Parameter
actual 135
compatibility 143
constant 137
ellipsis 143
formal 133
overview 133
typeless 138
UNIV 142
value 135
variable 136
Pascal
American National Standard 185
American National Standard. See Pascal, ANS.
ANS 20, 39, 170
Pascal User Manual and Report 20
PLFilePos 187
Pointer
function 54
qualifying 85
Pointers
assigning 54
creating 54
See also Types.
Precedence of operators 90

Pred function 44
Predefined numeric constants 27
Pree, Wolfgang 20
Preferences
    Language 178
private 153
Procedure
    as parameter 138
    declaration 126
    pointer 141
    statement 112
    type. See Types.
PROGRAM
    reserved word 24
PROTECTED 153
PUBLIC 153

Q

Qualifiers
    definition 81
    identification 25, 28
    pointers 110
    records 113
    variable 109

R

Real
    assignment 110
    type 43, 95
Records
    assignment 111
    in files 183
    type 58
    variant 59
Recursion 130
Relation function 100
Relational operators 99
REPEAT 107, 117
Reserved words
    case sensitivity 33
    definition 32
    list 32

S

SANE 100
Scalar
    assignment 110
    type 46
Scientific notation 34
Scope rules 164, 171
Self 157
Set
    assignment 110
    constructors 102
Sets
    See also Types.
    type 61
short-circuiting boolean expressions 97
SizeOf 27
SizeOf function 89
Statement
    CASE 108, 120
    FOR 107
    GOTO 108, 121
    IF 108, 118
    NULL 124
    overview 105
    repetition 106
    WHILE 106
String
    operators 97
Strings
    assignment 110
    changing length 53
    manipulating 52
    qualifying 82, 109
    storage 24
Strings. See Types.
Structured types
    See also Types.
Subranges
    See also Types.
    using parentheses 92
Succ function 45
Symbols
    definition 31
    special 32

T

Tag field identifier 60
text 184
The American Pascal Standard 20
TO 115
Trunc function 89
Turbo Pascal I/O Model 138
Types 64, 172
    arrays 56
    assignment compatibility 68
    BOOLEAN 48
    CHAR 49
    coercion 25, 69
    compatibility 66, 67
    declaration 70, 92, 163
    enumerated 49
    FILE 62
    floating point 43
    function 54
    INTEGER 46
    LONGINT 46
    object 63
    pointer 53
    procedure 54
    real 43
    record 58
    scalar 44
    set 61
    strings 51
    structured 55
    subrange 50
    UNSIGNEDLONG 48
    UNSIGNEDWORD 48

U

UNIV 25
Universal Interfaces
    SANE.p 100
UNSIGNEDLONG. See Types.
UNSIGNEDWORD. See Types.
UNTIL 106, 118
UpCase 28
USES 170, 173, 174, 178
Uses Propagation option 175, 176

V

VAR 133
Variables
    assembly, accessing 77
    assignment 109
    assignment compatibility 68
    C/C++, accessing 77
    declaration 76, 163
    qualifying 81
    using EXTERNAL 77

W

WHILE 116
Wirth, Niklaus 20
WITH 128

CodeWarrior

Pascal Language Reference

Credits

writing lead: Marc Paquette

other writers: Apple Computer, Inc., Benjamin Berman, John Roseborough

engineering: Marcel Achim, Nadine Bisson, Michael Cordery, Francois Plouffe

frontline warriors: Bill Catambay, Stephen Chong, James Derrick, Ron Levreault, Ron Liechty, Ray Pierce, Jim Trudeau, L. Frank Turovich, Jon Twigge, CodeWarrior users everywhere

Guide to CodeWarrior Documentation

CodeWarrior documentation is modular, like the underlying tools. There are manuals for the core tools, languages, libraries, and targets. The exact documentation provided with any CodeWarrior product is tailored to the tools included with the product. Your product will not have every manual listed here. However, you will probably have additional manuals (not listed here) for utilities or other software specific to your product.

Core Documentation	
IDE User Guide	How to use the CodeWarrior IDE
Debugger User Guide	How to use the CodeWarrior debugger
CodeWarrior Core Tutorials	Step-by-step introduction to IDE components
Language/Compiler Documentation	
C Compilers Reference	Information on the C/C++ front-end compiler
Pascal Compilers Reference	Information on the Pascal front-end compiler
Error Reference	Comprehensive list of compiler/linker error messages, with many fixes
Pascal Language Reference	The Metrowerks implementation of ANS Pascal
Assembler Guide	Stand-alone assembler syntax
Command-Line Tools Reference	Command-line options for Mac OS and Be compilers
Plugin API Manual	The CodeWarrior plugin compiler/linker API
Library Documentation	
MSL C Reference	Function reference for the Metrowerks ANSI standard C library
MSL C++ Reference	Function reference for the Metrowerks ANSI standard C++ library
Pascal Library Reference	Function reference for the Metrowerks ANS Pascal library
MFC Reference	Reference for the Microsoft Foundation Classes for Win32
Win32 SDK Reference	Microsoft's Reference for the Win32 API
The PowerPlant Book	Introductory guide to the Metrowerks application framework for Mac OS
PowerPlant Advanced Topics	Advanced topics in PowerPlant programming for Mac OS
Targeting Manuals	
Targeting BeOS	How to use CodeWarrior to program for BeOS
Targeting Java VM	How to use CodeWarrior to program for the Java Virtual Machine
Targeting Mac OS	How to use CodeWarrior to program for Mac OS
Targeting MIPS	How to use CodeWarrior to program for MIPS embedded processors
Targeting NEC V810/830	How to use CodeWarrior to program for NEC V810/830 processors
Targeting Net Yaroze	How to use CodeWarrior to program for Net Yaroze game console
Targeting Nucleus	How to use CodeWarrior to program for the Nucleus RTOS
Targeting OS-9	How to use CodeWarrior to program for the OS-9 RTOS
Targeting Palm OS	How to use CodeWarrior to program for PalmPilot
Targeting PlayStation OS	How to use CodeWarrior to program for the PlayStation game console
Targeting PowerPC Embedded Systems	How to use CodeWarrior to program for PPC embedded processors
Targeting VxWorks	How to use CodeWarrior to program for the VxWorks RTOS
Targeting Win32	How to use CodeWarrior to program for Windows